Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Rok Kralj

Paralelni in distribuirani algoritmi v numerični analizi

diplomsko delo

univerzitetni interdisciplinarni študijski program prve stopnje računalništvo in matematika

Mentor: doc. dr. Marjetka Krajnc

Ljubljana 2014

Besedilo je oblikovano s sistemom za pripravo besedil LaTeX.

Za nadzor nad spremembami tekom pisanja je bil uporabljen Git.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Paralelni in distribuirani algoritmi v numerični analizi

Kandidat naj razvije teoretično ozadje sočasnih sistemov in s tem povezanega aktorskega modela sočasnega izvajanja. Poudarek praktičnega dela diplomske naloge naj bo na distribuirani izvedbi algoritmov numerične analize in linearne algebre. Če se le da, naj algoritmi delujejo tudi na redkih matrikah. Kandidat naj vse opisane algoritme tudi realizira v enem izmed funkcijskih programskih jezikov z uporabo poljubnega ogrodja za izvedbo aktorskega modela.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Rok Kralj, z vpisno številko **63100261**, sem avtor diplomskega dela z naslovom:

Paralelni in distribuirani algoritmi v numerični analizi

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Marjetke Krajnc,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 12. septembra 2014 Podpis avtorja:

Zahvaljujem se svoji mentorici, doc. dr. Marjetki Krajnc, za njeno pomoč in strokovnost ter ves čas, ki mi ga je posvetila. Hvala tudi družini, prijateljem in vsem ostalim, ki so me vzpodbujali pri izdelavi tega diplomskega dela.

Kazalo

Seznam terminov

V spodnji tabeli se nahaja seznam pomembnejših terminov, uporabljenih v diplomskem delu.

angleško	**slovensko**	**kratica**
shared state	deljeno stanje	
scalability	razširljivost	
rendering	upodabljanje	
refactoring	preurejanje	
raytracing	sledenje žarkom	
pure function	čista funkcija	
load balancing	izenačevanje obremenitve	
local area network	lokalno omrežje	LAN
hash value	zgoščena vrednost	
garbage collector	sproščevalec pomnilnika	
finite state machine	končni avtomat	FSM
deadlock	smrtni objem	
critical section	kritični odsek	
application programming interface	programski vmesnik	API

Povzetek

Algoritmi numerične analize so široko uporabni v znanosti. Predvsem reševanje velikih sistemov enačb je stalnica znanstvenih simulacij na mnogih področjih, predvsem na področju fizike, strojništva, meteorologije in astronomije. Taki sistemi enačb so pogosto tako veliki, da jih zgolj z enim računalnikom ne moremo rešiti.

V diplomskem delu smo preučili in opisali numerične algoritme za računanje določenih integralov, množenje redkih in gostih matrik ter reševanje linearnih sistemov na distribuiranih arhitekturah. Še več, izvedli smo tudi prototip oblačne storitve za reševanje teh problemov s pomočjo aktorskega modela izračunavanja in ogrodja Akka.

Ključne besede: distribuirana arhitektura, aktorski model sočasnega izračunavanja, reševanje linearnega sistema enačb, množenje matrik, numerična integracija, funkcijsko programiranje, Scala, Akka.

Abstract

The algorithms studied in numerical analysis are widely used in science and engineering. They are especially common with large scientific simulations in many fields, ranging from astronomy and meteorology to physics and mechanical engineering. Systems of equations are commonly so big we are unable to solve them using just a single computer.

In this thesis we have examined the algorithms for numerical integration, both sparse and dense matrix multiplication and solving of systems of linear equations on a distributed memory machine. Moreover, we have also engineered a prototype of a cloud service for solving these kinds of problems using the actor model of concurrency.

Keywords: distributed architecture, actor model of concurrency, solving systems of linear equations, matrix multiplication, numerical integration, functional programming, Scala, Akka.

Poglavje 1

Uvod

Pri obravnavi numeričnih algoritmov pri numerični analizi se običajno zaradi lažjega razmišljanja o pravilnosti in učinkovitosti algoritma osredotočamo zgolj na njegovo matematično stran. Žal pa obstaja velik razkorak med idealnim, teoretičnim računskim strojem in med tem, kar imamo dejansko na voljo. Tako se morajo računalniki, na katerih te algoritme izvajamo, podrejati zakonom fizike in s tem vsem omejitvam, ki jih to prinaša.

Večina numeričnih algoritmov vsebuje vsaj eno zanko, ki ponavlja določen del algoritma. Eno tako ponovitev imenujemo ena iteracija. Na ta način zastavljen algoritem ni sam po sebi paralelen, saj ne moremo izvesti naslednje iteracije, ne da bi vedeli, kakšno spremembo je povzročila prejšnja. To za današnje računalnike pomeni, da jih s takim algoritmom ne moremo izkoristiti v celoti.

Želimo torej spremeniti algoritme na način, da problem razdelimo na več čim bolj neodvisnih enot, te enote razdeliti med razpoložljive računske vire in tako omogočiti sočasno izvajanje, nato pa njihove rezultate združiti v smiseln odgovor. Kako to naredimo in na kakšne probleme pri tem naletimo, pa bomo povedali v naslednjih poglavjih.

Končni rezultat diplomskega dela bo prototip oblačne storitve[1] za nekatere

[1] cloud computing

numerične algoritme. Sistem bo deloval na poljubni količini strežnikov, ki bodo med seboj ustrezno komunicirali. Omogočil bo študentom in delavcem fakultete, da po uspešni prijavi oddajo v izračun večje numerične probleme, ki jih sicer ne bi mogli rešiti zgolj z uporabo svojih osebnih računalnikov.

Poglavje 2

Sočasnost izvajanja

2.1 Motivacija

2.1.1 Velike količine podatkov

Steve Jobs, ustanovitelj podjetja Apple, je povedal, da ga je v začetkih podjetja zelo skrbelo, da bi kako drugo podjetje predstavilo hitrejši računalnik, kot jih je takrat izdeloval on. Zgodba pa se je popolnoma spremenila, ko je le nekaj let kasneje ustanovil podjetje Pixar, ki se je ukvarjalo s 3D digitalno grafiko in animacijo, delom, pri katerem lahko izkoristiš čisto vso računsko moč, ki jo imaš na voljo. Takrat pa je *močno* upal, da bi kdo izdelal hitrejši računski stroj, saj bi to zelo pospešilo razvojni proces ([Mat]).

Ni pa proizvodnja računalniške animacije edino področje, ki zahteva zmogljive računske stroje in sočasno izvajanje. Podoben problem so različne računalniške simulacije, od izračuna prenosov toplote do napovedovanja vremena, raziskovanje biološkega DNK in vesoljskih podatkov. Obstaja celoten segment tehnologij, ki se ukvarjajo s t.i. *big data* podatki. Pod ta dežnik spadajo ogromne podatkovne baze, od osrednjih registrov prebivalcev določene države, med drugim pa tudi veliki znanstveni izračuni, kot na primer podatki velikega hadronskega trkalnika v Cernu, ki zbere cel petabajt (10^{15} bajtov) podatkov vsako sekundo delovanja. Sem spadajo tudi spletni velikani, kot so

eBay, Google, Twitter in Facebook, saj ne le, da shranjujejo ogromne količine podatkov, ampak delajo na njih tudi obširne analize in izračune.

Kljub precejšnji specializiranosti tega področja pa uspe pisanje sočasne programske kode kaj kmalu postati domena povsem vsakdanjih programerjev, saj se napredek v zmogljivosti posameznega jedra počasi ustavlja.

2.1.2 Upočasnjevanje napredka

Empirično opažanje, da se število tranzistorjev v mikroprocesorju podvoji vsaki dve leti, imenujemo Moorov zakon, imenovan po Gordonu E. Mooru, soustanovitelju podjetja Intel. Človeštvo je dolga leta uživalo sadove takšne eksponentne rasti zmogljivosti računalniškega mikroprocesorja, žal pa se nezadržno bliža obdobje, ko nam bodo zakoni fizike dokončno stopili na prste.

Eksponentna rast ni več trajnostna, ko prestopimo meje abstraktnega. Vsi človeški izdelki so sestavljeni iz osnovnih gradnikov snovi, atomov, še več, vse fizikalne količine so kvantizirane, celo čas in dolžina [1]. Naravo eksponentne rasti je Gordon E. Moore osebno zelo lepo povzel v eni izmed svojih izjav: “It can’t continue forever. The nature of exponentials is that you push them out and eventually disaster happens.”

Za konec leta 2014 je Intel napovedal izid nove generacije mikroprocesorjev, imenovane Broadwell, osnovanih na 14 nm proizvodnem procesu. Intel je datum izida zaradi težav pri proizvodnji že enkrat zamaknil. Naj za primerjavo povemo, da je valovna dolžina rdeče svetlobe 700 nm, radij fluorovega atoma pa $0,1$ nm.

Vsa ta leta se je z večanjem števila tranzistorjev in frekvence procesorske ure povečevala zmogljivost procesorskega jedra. S takim povečevanjem zmogljivosti pridobijo čisto vse aplikacije. Zaradi problema odvajanja toplote od jedra in prej omenjenih omejitev so se proizvajalci preusmerili v proizvodnjo procesorjev z več jedri. Če želimo izkoristiti tak procesor, moramo temu posvečati posebno pozornost – pri nekaterih vrstah problemov pa je to popol-

[1] Planckov čas $5,39124 \cdot 10^{-44}$ sekunde. Planckova dolžina $1,616283 \cdot 10^{-35}$ metra

noma nemogoče.

Procesorjev s samo enim jedrom dandanes na trgu skoraj ni mogoče več najti. Glede na trenutni trend razvoja je očitno, da se bodo programerji enostavno morali naučiti pisati sočasno programsko kodo. Žal pa je to veliko težje, kot se zdi na prvi pogled. Nekaj s tem povezanih problemov in pasti bomo opisali v odseku 3.1.

2.2 Vrste sočasnih sistemov

V računalniških sistemih poznamo naslednje tipe arhitektur ([Čer10]):

- strogo zaporedne sisteme – sisteme **SISD** (angl. *single instruction single data*),
- paralelne sisteme – sisteme **SIMD** (angl. *single instruction multiple data*) in **MIMD** (angl. *multiple instruction multiple data*),
- distribuirane sisteme – omrežja odjemalec-strežnik, omrežja enakovrednih subjektov, ...

Strogo zaporedni sistemi so sistemi z enim procesorjem in pomnilnikom, ki tok ukazov izvaja strogo zaporedno. Tak sistem ni sočasen, temveč je zaporeden.

Paralelni sistemi so kompleksnejši, imajo več procesorskih enot s skupnim, tesno sklopljenim pomnilnikom. Taki sistemi se uporabljajo za procesorsko intenzivnejše aplikacije.

Porazdeljeni sistemi so bolj rahlo sklopljeni in navadno povezani preko lokalnega omrežja (LAN - *Local Area Network*) ali interneta, kjer si izmenjujejo sporočila. Glede na način komunikacije in organizacije porazdeljene sisteme naprej delimo na:

- sisteme odjemalec-strežnik (angl. *client-server*), kjer se med viri vrši asimetrična komunikacija, kar pomeni, da so posamezni subjekti bodisi

strežniki, ki strežejo povpraševanjem, bodisi odjemalci, ki povpraševanja izdajajo,

- omrežja enakovrednih subjektov (angl. *peer-to-peer networks, P2P*), kjer so subjekti popolnoma enakovredni in nastopajo v vlogi odjemalca in strežnika hkrati.

2.3 Dekompozicija problema

Ko smo soočeni z velikim problemom, ki se bo v prihodnosti poljubno povečeval, se moramo odločiti za primeren postopek delitve na manjše podprobleme. Pomembno je tudi, da taka delitev zadosti določenim orientacijskim kriterijem, ki nam lahko namignejo, ali je postopek delitve dober.

2.3.1 Načini dekompozcije

Oglejmo si nekaj splošnih možnosti dekompozicije problema:

- **Domenska subdivizija**, pri kateri razdelimo glavno geometrijsko domeno na več podobmočij. Dober primer uporabe take delitve je numerična integracija, kot bomo videli v poglavju 7.2. Zelo dober primer je tudi uporabljanje (angl. *rendering*) 3D grafike in filmov, ki se izvaja z metodo *raytracing* (sledenje žarkom), pri kateri lahko popolnoma različne koščke slike dodelimo različnim procesnim enotam z zelo malo medsebojne komunikacije.

- **Funkcionalna dekompozicija** (angl. *functional decomposition*), kjer glavni problem razdelimo na več neodvisnih komponent ali modulov. To srečamo pri mnogih ponudnikih storitev, kot so na primer spletne trgovine – ko uporabnik odda naročilo, lahko popolnoma vzporedno delujeta modul za plačevanje in modul za izračun predlogov izdelkov, ki bi kupca utegnili zanimati na podlagi njegovih preteklih naročil.

- **Paralelizacija polja** (angl. *array parallelism*) je na nek način primer domenske subdivizije, vendar jo omenjamo posebej, saj ima mnogo funkcijskih jezikov (med drugim tudi Haskell in Scala) vgrajene mehanizme, ki programerju omogočajo, da brez posebnega truda doda paralelnost v osnovne operacije (kot na primer `map`, `filter`, `find`, `sum`, `max`, `min`, `reduce`, ...) na poljih, vektorjih in matrikah.
- **Deli in vladaj** (angl. *divide and conquer*), kjer problem rekurzivno delimo na manjše podprobleme v drevesu podobnem načinu, dokler niso dovolj majhni, da jih reši ena sama procesna enota.
- **Cevovodno procesiranje** (angl. *pipelining*), kjer problem razdelimo na končno število zaporednih faz, vsako izmed faz pa je mogoče izračunati vzporedno.

Potrebno je poudariti, da zgoraj opisani načini niso edini mogoči, saj se včasih zgodi, da obstaja možnost za paralelnost že v sami (matematični) naravi problema, česar pa ne moremo uvrstiti v nobeno od zgornjih kategorij.

2.3.2 Želene lastnosti dekompozicije

Ko delimo glavni problem na več podproblemov, si želimo, da ima delitev naslednje lastnosti:

- da so podproblemi med seboj kar se da neodvisni. To pomeni, da naj bo med enotami, ki delujejo na različnih podproblemih, čim manj komunikacije,
- da je podproblemov številsko veliko več, kot je procesnih enot. To nam omogoča, da lažje razdelimo podprobleme med procesne enote na ta način, da ni nedejavnih enot, s tem pa je skupen čas izračunavanja minimalen,
- da so podproblemi vsaj približno enakomernih velikosti, saj lahko tako bolje razdelimo podprobleme procesnim enotam in bolje ocenimo čas do končanja opravila,

- da se z naraščanjem velikosti glavnega problema povečuje število podproblemov, ne pa njihova velikost.

2.4 Metrike sočasnega sistema

Naj bo T_1 čas, ki ga moramo počakati, da nam da rešitev sistem z eno procesno enoto, T_p pa čas izvajanja pri paralelni izvedbi s p procesnimi enotami. Potem lahko definiramo pohitritev take izvedbe:

$$S_p = \frac{T_p}{T_s}. \tag{2.1}$$

Rečemo, da je pohitritev linearna, če velja $S_p = p$. Praktično je to zelo težko doseči. Definirajmo še učinkovitost algoritma pri p procesnih enotah:

$$E_p = \frac{S_p}{p}.$$

2.4.1 Razširljivost

Razširljivost (angl. *scalability*) je sposobnost algoritma, da ustrezno deluje pri poljubnem povečevanju števila procesnih enot. Definicij, kdaj je algoritem razširljiv, je veliko, mi pa si bomo izbrali naslednjo:

Trditev 2.0.1 *Algoritem je razširljiv, če njegova učinkovitost* $E_p = \Theta(1)$ *ko* $p \to \infty$.

2.4.2 Amdahlov zakon

Imejmo p enako zmogljivih računskih enot. Naj faktor $B \in [0, 1]$ predstavlja delež algoritma, ki je strogo zaporeden, kar pomeni, da lahko na njem hkrati dela samo ena računska enota hkrati v preostalih $1 - B$ časa pa deluje vseh p. Tako je po formuli 2.1 celoten program za faktor

$$S_p = \frac{1}{B + \frac{1}{p}(1 - B)}$$

hitrejši. Za podan delež zaporednosti algoritma $B \in (0, 1]$ lahko izračunamo tudi maksimalno pohitritev, če pošljemo število računskih enot $p \to \infty$,

$$M = \lim_{p \to \infty} \frac{1}{p + \frac{1}{n}(1 - p)} = \frac{1}{B}. \tag{2.2}$$

Če ima nek algoritem $B = 0$, je tak algoritem **prijetno paralelen** (angl. *Pleasingly parallel*), njegov faktor maksimalne pohitritve (2.2) pa je neskončen.

Poglavje 3

Paralelni sistemi

Kot smo že omenili, procesorji ne postajajo hitrejši, ampak širši. To pomeni, da namesto, da naredimo eno računsko jedro bolj zmogljivo, več takih jeder združimo in jim omogočimo, da dostopajo do skupnega pomnilnika. Tako izvedeno procesorsko arhitekturo lahko izkoristimo bodisi tako, da na njej teče več programov hkrati (multi-tasking) ali več delov (niti) istega programa hkrati (to imenujemo multi-threading).

Predvsem v slednjem primeru, ko več niti istega programa rešuje isti problem, lahko to vodi do vseh vrst neugodnih situacij. Tudi ko več ljudi dela na istem problemu, se morajo zelo dobro uskladiti, drugače si stopajo po prstih – in pri algoritmih ni čisto nič drugače. Programiranje za več niti vodi do popolnoma nove skupine problemov.

3.1 Problemi deljenega pomnilnika

Za namene prikaza si poglejmo algoritem 1. Predstavlja enostaven prototip bankomata, ki omogoča dvig denarja, če je stanje na računu dovolj veliko.

Najprej definiramo spremenljivko `stanje` in jo nastavimo na $60,00$ €. To je količina denarja, ki ga naša banka dolguje svojemu komitentu. Definirajmo tudi proceduro, ki omogoča dvig tega denarja. Deluje tako, da najprej pre-

veri, če želeni znesek ne presega trenutnega stanja, nato izračuna novo stanje, izroči zahtevan denar komitentu, v zadnjem koraku pa zapiše prej izračunano preostalo stanje nazaj v spremenljivko `stanje`.

```
1  object Bankomat {
2     var stanje = 60.00
3
4     def dvigniDenar(koliko: Double) {
5        if (koliko <= stanje) {
6           val novoStanje = stanje - koliko
7           izrociBankovce(koliko)
8           stanje = novoStanje
9        } else {
10          opozori("Na računu imate premalo sredstev.")
11       }
12    }
13 }
```

Programska koda 1: Enostaven prototip bankomata.

Programska koda je na prvi pogled videti povsem pravilna. Sedaj pa si poglejmo, kaj se zgodi, če dve niti popolnoma vzporedno izvedeta dvig. Recimo, da prva nit izvede `Bankomat.dvigniDenar(50.00)` in druga nit `Bankomat.dvigniDenar(20.00)`. Pogoj

```
5        if (koliko <= stanje) {
```

je resničen s stališča obeh niti, zato lahko obe nadaljujeta. Takoj zatem se izvede prireditev

```
6           val novoStanje = stanje - koliko
```

kjer prva nit izračuna novo stanje 60,00 € - 50,00 € = 10,00 €, druga nit pa 60,00 € - 20,00 € = 40,00 €. Z ukazom

```
7           izrociBankovce(koliko)
```

najprej prva nit komitentu izroči 50,00 €, druga pa še 20,00 € – skupaj torej 70,00 €. V zadnjem, najbolj problematičnem koraku, pa najprej prva nit izvede

```
8           stanje = novoStanje
```

in tako v spremenljivko `stanje` zapiše 10,00 €, druga nit pa takoj za njo isto spremenljivko prepiše z vrednostjo 40,00 €. Očitno se je tu zgodil prepis iste spremenljivke v zelo kratkem časovnem razmiku.

Končni epilog je, da smo komitentu skupaj izročili 70,00 € ter mu na računu pustili še 40,00 €, čeprav je bilo njegovo začetno stanje na računu samo 60,00 €.

Izvajanje programske kode 1 na način, kot ni bil sprva predviden, je povzročilo dve hudi napaki. Kot prvo program ne bi smel dovoliti obeh dvigov; enega bi moral zaradi prenizkega stanja zavrniti. V nobenem primeru pa se ne bi smelo zgoditi, da je končno stanje 40,00 € namesto -10,00 €, saj lahko banka izgubi velike količine denarja, če ne odšteva izročenega denarja do centa točno.

Težava zgornje programske kode je t.i. **deljeno stanje** (angl. *shared state*). Deljeno stanje je kos pomnilnika (ali nek nabor spremenljivk), do katerega dostopa več niti hkrati. Spremenljivka `stanje` je idealen primer. Kadar imamo v programu deljeno stanje, moramo poskrbeti za pravilno zaklepanje. Da lahko to naredimo, nam dajo programski jeziki na voljo več različnih konstruktov. Najbolj enostaven izmed njih je *mutex*, s katerim lahko zaklenemo nek kos kode (imenovan kritični odsek – angl. *critical section*) na ta način, da ga lahko hkrati izvaja samo ena nit. Semafor (anlg. *semaphore*) ta privilegij dovoli n nitim. Naj na tem mestu izpostavimo še *readers-writers lock*, ključavnico, ki omogoča dostop bo nekega vira bodisi neomejenemu številu bralcev bodisi natanko enemu piscu.

Če pri zaklepanju nismo previdni, je lahko posledica smrtni objem (angl. *deadlock*), to je situacija, kjer prva nit zaklene vir, ki ga potrebuje druga nit in obratno, kar preprečuje, da bi katera koli izmed njiju nadaljevala. Še več,

če zaklepamo preveč na grobo (angl. *coarse-grained*), je rezultat lahko slabša izkoriščenost računskih virov.

Programiranje na tak način je težko. Empirično so programske napake zaradi večnitne programske kode ene izmed najtežjih napak za odkrivanje, saj se napaka ne pojavi nujno vsakič in ne nujno takoj. Poznani so primeri iz prakse, kjer je celoten sistem uspešno tekel več mesecev, preden se je sesul zaradi s strani programerjev nepredvidenih okoliščin. Zaradi kompleksnosti interakcij med nitmi je odprava takih napak zelo zamudna in draga.

V tej diplomski nalogi se bomo v celoti izognili uporabi deljenega stanja in s tem vsem problemom, ki jih prinaša.

Poglavje 4

Distribuirani sistemi

Distribuiran sistem je sistem, ki je sestavljen iz več procesnih enot, vsake s svojim pomnilnikom. Eno tako enoto v žargonu distribuiranih arhitektur poimenujemo tudi *proces*. Procesi med seboj komunicirajo s pošiljanjem sporočil (angl. *message passing*) preko omrežja neke vrste, pogosto je to kar lokalno omrežje (LAN), lahko pa je tudi internet.

Ker pomnilnik ni deljen, je tak sistem bolj odporen na probleme, ki smo jih omenili prej, ima pa kar nekaj drugih posebnosti.

4.1 Posledice omrežne komunikacije

Če želimo komunicirati preko omrežja, se moramo zavedati nekaterih zelo pomembnih razlik.

Ker gre pri omrežni komunikaciji v najbolj grobem smislu za prenos zaporedja bitov iz ene lokacije na drugo, moramo zagotoviti primerne serializacijske algoritme. Podatkovna struktura, ki jo želimo poslati, je lahko razkropljena po celotnem pomnilniku računalnika, zato jo je potrebno prej zbrati in sestaviti v dogovorjeno obliko. Obraten proces, deserializacija, poteka na prejemnikovi strani.

Pri komunikaciji prek omrežja, sploh interneta, imamo na voljo veliko

manjšo pasovno širino, zato se želimo izogniti vsakemu odvečnemu prenašanju podatkov. Vlogo pri neželenem upočasnjevanju komunikacije igra tudi latenca, torej časovni zamik med začetkom pošiljanja in trenutkom, ko prejemnik prejme prvi kos sporočila. Latenca ni odvisna od velikosti sporočila.

Pri omrežni komunikaciji obstaja tudi večja možnost, da se sporočilo izgubi ali pa prispe na cilj poškodovano. Za odpravo te vrste težav je zadolžen protokol TCP, saj za vsak poslani mrežni paketek zahteva potrditev o prejemu (ACK) s strani prejemnika, skupaj s podatki pa pošlje tudi zgoščeno vrednost (angl. *hash value*) in na ta način omogoči prejemniku, da se prepriča, da je sporočilo res prispelo nepoškodovano.

4.1.1 Želene lastnosti komunikacije

Čas, ki ga potrebujemo za pošiljanje sporočila preko omrežja, sploh interneta, se v človeških merilih zdi zanemarljivo kratek, zato je človeški vidik zavajajoč. V tabeli 4.1 si lahko za občutek pogledamo te čase v 1.000.000.000-krat večjem merilu. Tudi zato je pomembno, da je komunikacija med procesi čim bolj učinkovita. Nekaj namigov, kako to dosežemo:

- minimiziramo pogostost komuniciranja,
- minimiziramo količino poslanih podatkov,
- poskušamo enakomerno obremeniti vse komunikacijske kanale,
- poskrbimo, da se pošiljanje čim bolj prekriva z izračunavanjem.

Najbolj pomembna točka je zadnja. Vsa komunikacija, ki se izvaja hkrati z izračunavanjem je s praktičnega vidika gledano, zastonj, saj ne doda ničesar k skupnemu trajanju opravila. Če smo še posebej spretni in narava problema to omogoča, lahko čisto vso komunikacijo izvedemo na tak način in tako ves čas izkoriščamo čisto vse računske vire.

Vse to nam bo omogočil aktorski model, saj podpira asinhrono komunikacijo, kar pomeni, da se pošiljanje delegira drugi enoti v sistemu, aktor pa takoj nadaljuje z izračunavanjem. Več o tem bomo povedali v odseku 5.2.

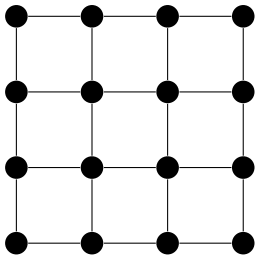
Dododek v računalniškem sistemu	Približno trajanje	Primerjava
Izvedba procesorskega ukaza	1 ns	1 sekunda
Dostop do L1 predpomnilnika	0,5 ns	0,5 sekunde
Napačna napoved veje	5 ns	5 sekund
Dostop do L2 predpomnilnika	7 ns	7 sekund
Odklepanje/zaklepanje mutexa	25 ns	25 sekund
Dostop do delovnega pomnilnika (RAM)	100 ns	1,5 minute
Pošiljanje 2KB preko 1Gb/s omrežja	20.000 ns	5,5 minut
Branje 1MB iz delovenga pomnilnika (RAM)	250.000 ns	70 minut
Premik bralne glave trdega diska (HDD)	8.000.000 ns	92 dni
Branje 1MB s trdega diska (HDD)	20.000.000 ns	8 mesecev
Pošiljanje paketka iz Amerike v Evropo in nazaj	150.000.000 ns	5 let

Tabela 4.1: Prikaz trajanj različnih dogodkov v človeškem merilu.

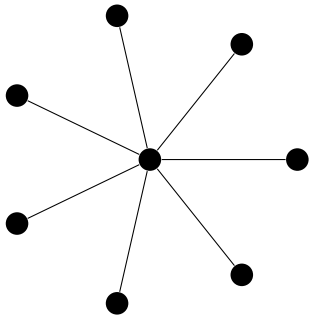
4.2 Topologija omrežja

Odvisno od vrste problema, ki ga rešujemo, in tudi od uporabljenega algoritma, se lahko odločimo, na kakšen način bomo razporedili in povezali procese med seboj. Poglejmo si nekaj možnosti.

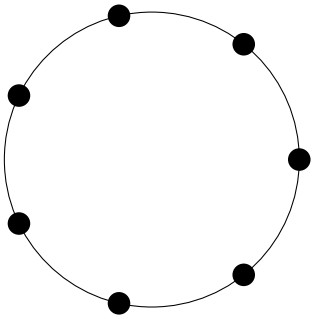
Slika 4.1: Topologija črte.



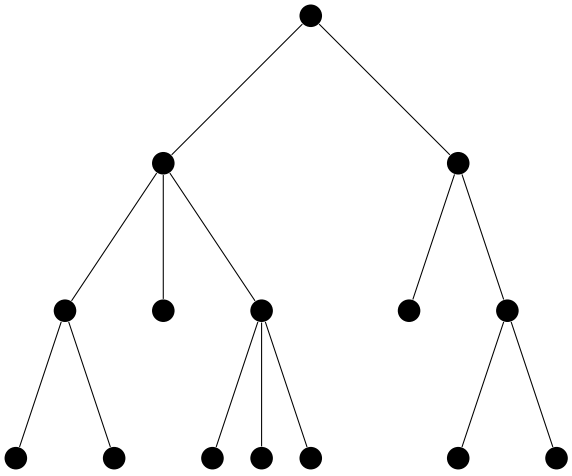
Slika 4.2: Topologija mreže.



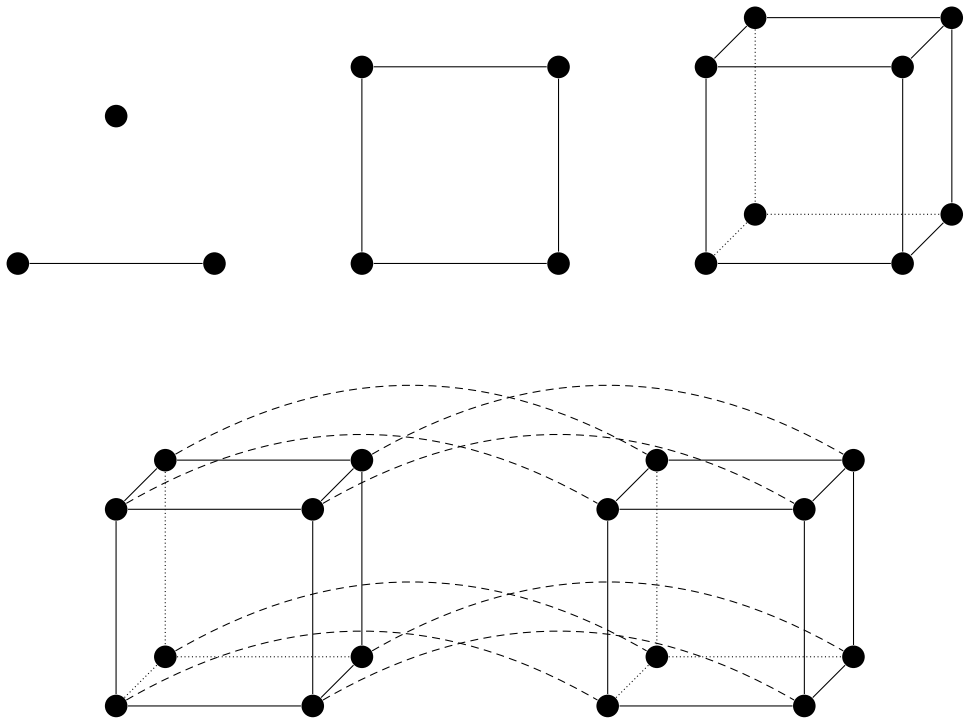
Slika 4.3: Topologija zvezde.



Slika 4.4: Topologija obroča.



Slika 4.5: Topologija drevesa.



Slika 4.6: Topologija hiperkocke (primeri za več dimenzij).

Pri izdelavi sistema je pomembno tudi to, da znamo pravilno odreagirati v primeru odpovedi določenega procesa. Odpoved centralnega vozlišča pri topologiji zvezde (slika 4.3) lahko pomeni odpoved celotnega sistema, medtem ko ob odpovedi največ enega procesa v topologiji mreže (slika 4.2) oz. največ $d-1$ procesov v primeru d-dimenzionalne hiperkocke (slika 4.6) graf še vedno ostane povezan.

4.3 Izenačevanje obremenitve

Izenačevanje obremenitve (angl. *load balancing*) je izraz za sposobnost sistema, da se prilagaja obremenitvam in tako obremeni procese čim bolj enakomerno. Če en proces neenakomerno obremenimo, bo skupni čas reševanja problema daljši.

Jasno pa je, da izenačevanje obremenitve ni čisto zastonj, sploh pa ne

pri distribuiranih arhitekturah, saj je potrebno precej komunikacije za sporočanje trenutne obremenjenosti posameznega procesa. Poznamo dve vrsti izenačevanja obremenitve:

- Statično izenačevanje obremenitve poteka, ne da bi morali procese povpraševati o njihovi obremenjenosti. Predstavljajmo si, da imamo 15 podproblemov in 3 procese. Lahko bi prvemu dodelili podprobleme od 1 do 5, drugemu od 6 do 10, tretjemu pa še preostalih 5. Predvsem za tak način izenačevanja je pomembno, da so podproblemi podobnih velikosti in procesi podobnih zmogljivosti.

- Dinamično izenačevanje obremenitve poteka v sodelovanju s procesi. Tu se odločamo kateremu procesu bomo dodelili naslednji podproblem glede na to, kdo je najmanj obremenjen ali pa kdo ima najmanj podproblemov nakopičenih v čakalni vrsti. Če so latence dolge, se lahko zgodi, da tak način izenačevanja zmogljivosti upočasni sistem do te mere, da je statično izenačevanje boljše.

Poglavje 5

Aktorski model

Kot smo že omenili v odseku 3.1, je pisanje paralelne programske kode s standardnimi prijemi (nitmi in ključavnicami) zelo težko. S tem v mislih je smiselno, da izberemo nek drug način izkoriščanja več procesnih enot.

Aktorski model (ang. Actor Model) je na matematičnih konceptih osnovan model izračunavanja, ki omogoča, da je paralelna programska koda lažje razumljiva in bolj modularna, predvsem pa brez programskih napak, saj je o njej dosti lažje analitično razmišljati.

Osnovna ideja tega računskega modela sega v leto 1973, inspiracija zanj pa prihaja iz osnovne človeške interakcije, vsebuje pa tudi precej fizikalnih idej. Glavna prednost tega modela je ta, da odpravi prej omenjeno *deljeno stanje* (angl. *shared state*), torej v grobem pomenu kos pomnilnika (oz. nek nabor spremenljivk), do katerega v kratkem časovnem obdobju dostopa več niti. Namesto tega izračunavanje začnemo s končnim številom neodvisnih entitet, imenovanih aktorji. Entitete si med seboj ne delijo ničesar, dovoljeno pa jim je komuniciranje med sabo samo s pošiljanjem sporočil.

Zaradi svoje naravne podobnosti medčloveški interakciji je aktorski model zelo primeren za modeliranje vseh vrst sistemov, ki se pojavljajo v vsakdanjem življenju. Zaradi svojih lastnosti je Aktorski model zelo primeren za izvedbo sistema končnih avtomatov (angl. FSM - *finite state machine*).

Aktorski model je v zadnjih nekaj letih močno pridobil na priljubljenosti, zato zanj obstaja vse več knjižnic in ogrodij. Na tem mestu naj izpostavimo dva, in sicer Cloud Haskell (knjižnica za programski jezik Haskell) in Akka (ogrodje za programska jezika Java in Scala). Uporabili bomo slednjega.

5.1 Aktor

Aktor (po [Hew14]):

1. je neodvisna entiteta z identiteto,
2. ima obnašanje,
3. lahko z ostalimi aktorji komunicira samo preko pošiljanja sporočil (angl. *message passing*).

Aktor ima svojo identiteto, torej svoje obnašanje in notranje stanje, do katerega lahko dostopa samo on. Tako kot ljudje nimamo sposobnosti, da bi drug drugemu brali misli, tudi aktorji ne morejo neposredno dostopati do notranjega stanja drugega aktorja. Lahko ga le prosijo, da jim ta želene podatke posreduje.

Aktor ima obnašanje. To pomeni, da lahko, ko prejme sporočilo, odreagira tako, da hkratno izvede katerokoli (ali več) spodnjih akcij:

- pošlje končno število sporočil ostalim aktorjem, katerih naslove pozna,
- ustvari končno število novih aktorjev,
- spremeni obnašanje do naslednjega sporočila (spremeni svoje notranje stanje).

Eno izmed najbolj pomembnih zagotovil aktorskega modela je, da se vsak aktor ukvarja samo z enim sporočilom hkrati. Če je aktor zaposlen, so sporočila zanj dostavljena v njegov poštni nabiralnik, aktor pa jih bo v vrstnem redu prihoda (FIFO) obdelal takoj, ko bo spet prost.

To omogoča programerju, ki programira navodila za obnašanje posameznega aktorja, da pozabi na vse nevarnosti in tegobe pisanja večnitne programske kode.

5.2 Komunikacija med aktorji

Vsa komunikacija med aktorji poteka asinhrono, kar pomeni, da aktor sporočilo nemudoma pošlje in ne čaka na uspešno pošiljanje, kaj šele na morebitni odgovor, ampak takoj nadaljuje z delom. Zelo pomembno je tudi, da aktorje programiramo na ta način, da nikoli po nepotrebnem ne zasedajo niti s čakanjem (so neblokirajoči). To med drugim omogoča, da lahko na povprečnem računalniku živi tudi do nekaj miljonov aktorjev hkrati.

Z namenom medsebojne komunikacije vsakemu aktorju pripada naslov, vendar ga poznajo samo nekateri aktorji. Tak način omejevanja komunikacije je idealen za gradnjo različnih vrst omrežnih arhitektur. Nove aktorje ustvarjajo že obstoječi aktorji, zato naravno tvorijo hierarhično (drevesno) topologijo, vendar je nismo prisiljeni uporabljati, če je ne želimo.

5.3 Lokacijska neodvisnost

Aktorski model je bil zgrajen z lokacijsko neodvisnostjo v mislih. To pomeni, da za programerja ni pomembno, kje aktor živi. Naj bo to na istem računalniku, v računalniškem centru, ali pa na drugi celini; v vsakem primeru bo komunikacija s takim aktorjem potekala popolnoma enako, le latence bodo verjetno daljše. To je zelo priročno v razvojni fazi, saj imamo lahko celotno arhitekturo postavljeno na enem samem računalniku, nato pa jo ob koncu projekta brez posebnega truda realiziramo.

Z namenom lokacijske neodvisnosti nam ogrodja običajno ponudijo referenco (v ogrodju Akka je to na primer `ActorRef`), ki vsebuje vse potrebno, da pridemo v stik z aktorjem.

Poglavje 6

Uporabljene tehnologije

6.1 Programski jezik Scala

Scala je relativno nov objektno-funkcijski programski jezik, saj se je pojavil šele leta 2003. Beseda Scala je sestavljenka angleških besed **sca**lable in **la**nguage, kar simbolizira sposobnost jezika, da raste skupaj z zahtevami svojih uporabnikov ([OSV11]).

Teče na Javanskem stroju (JVM - Java Virtual Machine), kar programerju omogoča, da uporablja celoten nabor knjižnic in infrastrukture, ki je na voljo za programski jezik Java. Posledica uporabe Javanskega stroja je tudi hitrost izvajanja programov, plod dolgoletnega razvoja in izboljšav.

V Scali je vsaka vrednost objekt in vsaka operacija na njem je klic metode. Izraz `1 + 2` je primer klica metode, imenovane `+`, na celem številu 1, kar bi lahko povsem legitimno zapisali kot `1.+(2)`. Tudi funkcije so v Scali objekti.

Jezik je močno in statično tipiziran, kar omogoča programerjem, da programske napake odkrijejo že v fazi prevajanja, še preden je program prvič pognan.

6.1.1 Funkcijski programski jeziki

Scala je poleg svoje objektne orientacije tudi zmogljiv funkcijski programski jezik. Ideje o funkcijskem programiranju so starejše od prvih elektronskih računalnikov, temeljijo pa na lambda računu, osnovanem v začetku tridesetih let prejšnjega stoletja. Prvi funkcijski jezik, Lisp, se je pojavil davnega leta 1958. Ostali funkcijski programski jeziki, ki so še danes precej priljubljeni, so Scheme, SML, Erlang, Haskell, OCaml in F# ([OSV11]).

Dolgo časa je bilo funkcijsko programiranje zgolj domena akademskih institucij, vendar je v zadnjih nekaj letih opaziti precej elementov funkcijskega programiranja v veliko povsem vsakdanjih programskih jezikih. Naj na tem mestu omenimo zgolj Javo 8 ter komponento LINQ programskega jezika C#.

V funkcijskih jezikih je programiranje dosti bolj osredotočeno na to, *kaj* želimo izračunati, kot pa *kako* in v kakšnem zaporedju to narediti. Če za nek jezik rečemo, da je funkcijski, imamo v mislih predvsem tri stvari: jezik obravnava funkcije popolnoma enako kot vsak drug podatek, podatkovne strukture so nespremenljive (angl. *immutable data structure*) in funkcije so po svoji naravi čiste (angl. *pure function*).

Funkcije so "prvorazredni državljani"

Programski jezik funkcije obravnava popolnoma enakopravno ostalim podatkovnim tipom, kot na primer nizom ali številom. To pomeni, da je funkcije mogoče prirediti konstantam, jih podati kot parametre drugim funkcijam (take funkcije imenujemo funkcije višjega reda[1]), jih iz funkcij vračati, zgraditi polje (angl. *array*) funkcij, izračunati kompozitum več funkcij,...

Funkcijo lahko definiramo, ne da bi jo morali prej poimenovati[2] – povsem enako kot imamo lahko v izrazu številske vrednosti zapisane neposredno, brez imena.

[1] na primer integral in odvod

[2] anonimna funkcija (angl. *anonymous function*)

Nespremenljive podatkovne strukture

Čisto tako kot je to običajno v matematiki, se vrednosti že določenih spremenljivk (torej konstant) ne spreminjajo. Kadar pišemo algoritem v matematični psevdokodi, namesto da v vsaki iteraciji zanke prepišemo staro vrednost spremenljivke x z novo, jo shranimo v spremenljivko z za ena višjim indeksom (x_{i+1}). Funkcijski jeziki delujejo na enak način.

Ko podatek nekam zapišemo, ga ne smemo več popravljati, niti ga ne smemo pobrisati. Za izbris podatkov je namesto nas zadolžen sproščevalec pomnilnika (angl. *garbage collector*), ki sproti briše podatke, za katere ve, da jih ne bomo več potrebovali, deluje pa povsem samodejno.

Razmišljanje o delovanju takega programa je dosti bolj enostavno. Če neka nit drži kazalec na nek podatek, z gotovostjo ve, da bo ta podatek tam tudi čez poljubno mnogo časa – še več, podatek bo točno tak, kot je bil. Torej tudi ni strahu, da bi druga nit podatek med izvajanjem prve niti spreminjala. S takim načinom programiranja lahko zaobidemo skoraj čisto vse probleme, omenjene v odseku 3.1.

Ni pa tak način dela čisto brez slabih strani. Če želimo v nekem nizu znakov popraviti samo eno črko, moramo pogosto celoten niz prekopirati na drugo pomnilniško lokacijo, kar je zelo potratno, vendar smo le tako lahko prepričani, da bo imela morebitna druga nit še vedno na voljo staro različico.

Naj še omenimo, da z uporabo stalnih podatkovnih struktur (angl. *persistent data structures*, torej takih, ki hranijo vse prejšnje različice) kopiranje celotne predhodne strukture ni nujno potrebno. Pri dodajanju elementa v množico (`Set`) bo tako na primer nova nit hranila na novo narejeno spremembo in kazalec na prejšnjo različico.[3]

[3]Dejanska izvedba običajno ustvari novo podatkovno strukturo, katere deli kažejo na tiste dele prejšnje, ki so ostali enaki.

Čiste funkcije

Funkcijski programski jezik programerja spodbuja (včasih celo prisili), da piše funkcije brez stranskih učinkov, kot je to povsem samoumevno v matematiki. To pomeni, da funkcija zgolj sprejme parametre in vrne svoj rezultat, hkrati pa ne povzroči dogodkov, kot so recimo izpis na zaslon, brisanje datoteke, pošiljanje elektronske pošte in podobnih.

Program bo tako deloval povsem enako, če funkcijo namesto enkrat izvedemo večkrat ali pa je sploh ne. Kako se lahko to zgodi? Imejmo enostavno funkcijo `vsota(a: Int, b: Int)`, ki sešteje dve celi števili, definirano takole:

```
1 def vsota(a: Int, b: Int) = a + b
```

Če se sedaj nekje v programu nahaja klic `vsota(3, 2)`, bo vsak razumno zmogljiv prevajalnik opazil, da lahko program enostavno pohitrimo tako, da klic funkcije popolnoma zamenjamo kar z njenim rezultatom (`5`), saj je znan vnaprej, še preden program prvič poženemo. Če bi imela funkcija `vsota` kakršenkoli stranski učinek, se ta v primeru take zamenjave ne bi zgodil.

Če želimo, da je funkcija čista, mora poleg vsega zgoraj omenjenega veljati še, da je rezultat funkcije odvisen samo od njenih parametrov. To pomeni, da na rezultat ne vplivajo nobeni zunanji dejavniki, kot je to recimo trenutni čas, generator naključnih števil, morebitni internetni promet, datoteke na uporabnikovem datotečnem sistemu,...

Lastnost, ko nam je dovoljeno klic funkcije zamenjati z njegovim rezultatom, ne da bi spremenili pomen programa, se imenuje **referenčna transparentnost** (angl. *referential transparency*).

6.1.2 Prednosti Scale

6.1.3 Jedrnata programska koda

Programi v programskem jeziku Scala so pogosto zelo kratki, kar pomeni ne samo manj tipkanja, ampak tudi manj napora pri branju in manj priložnosti,

da se kam skrije kak programski hrošč.

```
class Avtomobil {
   private String ime;
   private int kilometrina;

   public Avtomobil(String ime, int kilometrina) {
      this.ime = ime;
      this.kilometrina = kilometrina;
   }

   public String getIme() {
      return ime;
   }

   public int getKilometrina() {
      return kilometrina;
   }
   public void setKilometrina(int kilometrina) {
      this.kilometrina = kilometrina;
   }
}
```

Programska koda 2: Definicija razreda Avtomobil v programskem jeziku Java, ki dovoli, da mu nastavljamo kilometre.

Programska koda 2 prikazuje ustvarjanje razreda `Avtomobil`, ki objektom dovoli, da jim nastavljamo kilometre, dokler pustimo ime pri miru. V Scali je mogoče podobno funkcionalnost doseči z zgolj eno vrstico, kot to prikazuje programska koda 3.

```
class Avtomobil(val ime: String, var kilometrina: Int)
```

Programska koda 3: V Scali je mogoče razred Avtomobil definirati brez nepotrebnega pisanja.

Programiranje na visokem nivoju

Scala omogoča programerju, da programira na višjem nivoju abstrakcije ([OSV11]).

Imejmo niz znakov, shranjen v spremenljivki `ime`, programer pa bi rad ugotovil, če vsebuje vsaj eno veliko črko. To lahko v programskem jeziku Java storimo s programsko kodo 4.

```
1 boolean vsebujeVelikoCrko = false;
2 for (int i = 0; i < ime.length(); ++i) {
3    if (Character.isUpperCase(ime.charAt(i))) {
4       vsebujeVelikoCrko = true;
5       break;
6    }
7 }
```

Programska koda 4: Košček programske kode v jeziku Java, ki ugotovi, če besedilo vsebuje vsaj eno veliko črko.

Takšno programiranje je precej zamudno – še več, programer, ki bo program vzdrževal, bo potreboval več časa, da ugotovi, kaj ta programska koda sploh počne. V Scali je isto mogoče zapisati krajše in bolj pregledno.

```
1 val vsebujeVelikoCrko = ime.exists(_.isUpper)
```

Programska koda 5: Primerljiva programska koda v jeziku Scala, ki preveri, če niz vsebuje vsaj eno veliko črko.

Kot smo že omenili – ko programiramo v Scali, se bolj osredotočamo na to, *kaj* želimo narediti, ne pa toliko na to, *kako* to naredimo.

Močna tipizacija programskega jezika

Kot smo že omenili, je Scala močno tipiziran jezik, kar prinaša cel kup prednosti, predvsem pa daje večje zagotovilo, da je program pravilen in olajša preurejanje (angl. *refactoring*) že napisane programske kode. Poleg tega je močno tipizirana koda na nek način dokumentacija sama zase, v kombinaciji z

modernim razvojnim okoljem (IDE) pa ponuja razvojno izkušnjo na visokem nivoju.

Po drugi strani pa tak način pisanja programske kode zahteva, da je na vsaki točki programa za vsako spremenljivko točno določen tudi njen podatkovni tip. To je manj praktično, saj mora programer ob vsaki spremenljivki eksplicitno navesti tudi njen podatkovni tip, kar povzroči daljšo in manj berljivo programsko kodo. To je razlog, zakaj ima veliko programerjev raje šibko tipizirane jezike – takšno programiranje se jim zdi enostavno preveč zamudno.

Scala tega ne zahteva, ampak programerju ponudi srednjo pot; sposobna je namreč izračunati podatkovni tip iz vhodnih parametrov skoraj brez žrtvovanja pozitivnih lastnosti močne tipizacije. Na ta način nam znatno olajša programiranje. Poglejmo si primer.

```
1 val poved = "Ta poved vsebuje več besed"
2 val dolzine = poved.split(" ").map(_.length)
```

Programska koda 6: Scala je sposobna izračunati podatkovni tip podatka iz vhodnih parametrov.

Očitno je konstanta `poved` tipa `String`, torej niz znakov. V konstanti `dolzine` se po izvedbi programa nahaja `Array(2, 5, 7, 3, 5)`. Scala je brez pomoči programerja sposobna ugotoviti, da je podatkovni tip te konstante `Array[Int]`.

6.2 Ogrodje Akka

Ogrodje Akka je odprtokodno ogrodje, ki nam olajša izvedbo paralelnih in distribuiranih arhitektur s pomočjo prej omenjenega aktorskega modela. Ogrodje teče na Javanskem virtualnem stroju, podpira pa programska jezika Java in Scala.

Ogrodje je zgrajeno s poudarkom na hitrosti in učinkovitosti. Avtorji trdijo, da je mogoče shraniti 2,5 milijona aktorjev na gigabajt pomnilnika,

aktorji pa lahko komunicirajo s hitrostjo tudi do 50 milijonov sporočil na sekundo na enem samem računalniku.

6.3 Knjižnica Breeze

Breze je hitra in zmogljiva odprtokodna knjižnica za numerično procesiranje. Vsebuje predvsem matrične in vektorske operacije. V ozadju uporablja zbirko funkcij BLAS[4] in knjižnico LAPACK[5], če ju imamo nameščene. Uporaba teh knjižnic omogoča zelo hitre operacije, optimizirane pa so tudi za učinkovito izrabo procesorskih predpomnilnikov.

6.3.1 BLAS

BLAS je zbirka funkcij za osnovne matrične in vektorske operacije v linearni algebri. Njene izvedbe so tako zelo učinkovite, da so jih uporabili celo avtroji programskih paketov MATLAB in Mathematica. Funkcije so razdeljene na tri nivoje:

- V prvem nivoju se nahajajo operacije, ki vključujejo zgolj vektorje in skalarje. Sem spada izračun skalarnega produkta ($x^T y$) in kvadratne norme ($||x||_2$), razteg vektora (αx), razteg vektorja v kombinaciji s premikom ($\alpha x + y$), rotacije vektorja ter še nekaj drugih.

- Drugi nivo vsebuje matrično-vektorske operacije, kot na primer $\alpha Ax + \beta y$, vsebuje pa tudi reševanje sistema $Tx = y$, če je T trikotna matrika.

- V tretjem, zadnjem nivoju se nahajajo matrično-matrične operacije, kot je na primer najbolj splošno množenje in seštevanje treh matrik ($\alpha AB + \beta C$).

[4]Basic Linear Algebra Subprograms
[5]Linear Algebra PACKage

6.3.2 LAPACK

LAPACK je matrična knjižnica, zadolžena za naprednejše matrične operacije, torej iskanje lastnih vrednosti, reševanje sistema linearnih enačb in razcepe matrike, kot so na primer LU in QR razcep, razcep Choleskega ter Schurova dekompozicija. Zanimivo pri tej knjižnici je, da matrike razdeli v 28 razredov, določitev algoritma za želen problem pa je odvisna od tega, v kateri razred spadajo vhodne matrike. Razredi matrik so označeni z dvočrkovno oznako, vključujejo pa diagonalne matrike (`DI`), ortogonalne matrike (`OR`), unitarne matrike (`UN`), simetrične matrike (`SY`), zgornje Hessenbergove matrike (`HS`), simetrične ali pozitivno definitne matrike (`PO`), itd. Če matrika v splošnem nima nobenih posebnih lastnosti, jo uvrstimo v razred splošnih matrik (`GE`) in se s tem odpovemo morebitnim pohitritvam na podlagi matematičnih dognanj. Knjižnica bi si zaradi svoje zanimive narave zaslužila obširnejši opis, a je to žal izven okvira tega diplomskega dela.

Sintaksa knjižnice Breeze in poimenovanje metod se zgleduje po programskemu paketu MATLAB, med drugim pa so avtorji dodali kar nekaj funkcionalnosti, ki olajšajo delo. Ena takih je na primer sposobnost ustvarjanja nove matrike preko funkcije vrstice in stolpca. Spodnji košček kode ustvari Vandermondovo matriko[6] dimenzije 5×5.

```
1 val x = DenseVector(2.0, 5.0, 7.0, 4.0, 12.0);
2
3 DenseMatrix.tabulate(5,5) { case (v, s) =>
4    Math.pow(x(v), s)
5 }
```

[6]Vandermondova matrika pripadajočemu vektorju x je kvadratna matrika, ki ima v $(v \in \mathbb{N})$-ti vrstici in $(s \in \mathbb{N})$-tem stolpcu element $(x_v)^{s-1}$.

Poglavje 7

Numerična integracija

7.1 Matematično ozadje

7.1.1 Newton-Cotesova integracijska pravila

Newton-Cotesova integracijska pravila so družina formul za numerično integracijo, osnovana na evaluaciji funkcije na ekvidistantnih točkah.

Radi bi izračunali integral

$$I(f) = \int_a^b f(x)\,dx,$$

kjer je funkcija f definirana na intervalu $[a, b]$. Imejmo tudi $n+1$ ekvidistantnih točk x_i, definiranih takole:

$$h = \frac{b-a}{n},$$
$$x_i = a + ih; \quad i = 0, \ldots, n.$$

Naj bo $y_k = f(x_k)$. Ločimo dva tipa Newton–Cotesovih pravil ([Ple10]):

- zaprti tip: $\int_a^b f(x)\,dx = \sum_{k=0}^{n} A_k y_k + R_n(f)$,
- odprti tip ($n \geq 2$): $\int_a^b f(x)\,dx = \sum_{k=1}^{n-1} B_k y_k + R_n(f)$,

pri čemer smo z $R_n(f)$ označili napako pravila.

Nekaj osnovnih zaprtih pravil

- Pri $n = 1$ dobimo *trapezno pravilo*:

$$\int_{x_0}^{x_1} f(x)\,dx = \frac{h}{2}(y_0 + y_1) - \frac{h^3}{12}f''(\xi),$$

kjer je $\xi \in [x_0, x_1]$. Kot je zaznati že iz imena, smo funkcijo aproksimirali s premico, ki skupaj z abscisno osjo na danem intervalu tvori trapez, čigar ploščino vzamemo za približek integrala funkcije. Uteži

$$A_0 = \int_{x_0}^{x_1} \frac{x - x_1}{x_0 - x_1}\,dx = \frac{h}{2}$$
$$A_1 = \int_{x_0}^{x_1} \frac{x - x_0}{x_1 - x_0}\,dx = \frac{h}{2}$$

smo izračunali preko integralov Lagrangeevih baznih polinomov. Za napako velja

$$\begin{aligned} R_1(f) &= \int_{x_0}^{x_1} \frac{f''(\xi_x)}{2}(x - x_0)(x - x_1)\,dx = \\ &= \frac{f''(\xi)}{2}\int_{x_0}^{x_1} (x - x_0)(x - x_1)\,dx = \\ &= -\frac{h^3}{12}f''(\xi), \end{aligned}$$

Pri izpeljavi napake smo uporabili izrek o povprečni vrednosti, saj je polinom $(x - x_0)(x - x_1)$ konstantnega predznaka na $[x_0, x_1]$.

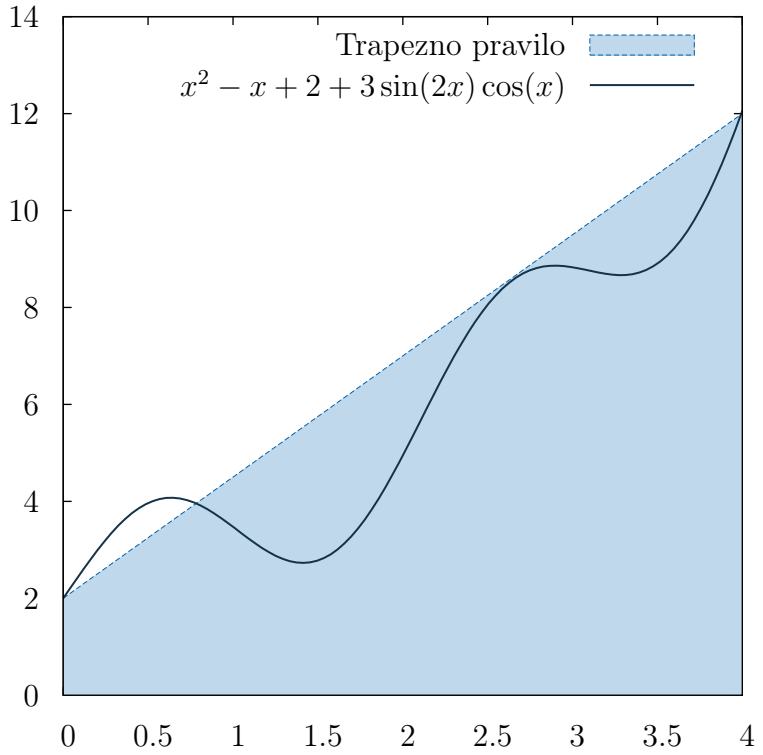
- Pri $n = 2$ dobimo *Simpsonovo pravilo*:

$$\int_{x_0}^{x_2} f(x)\,dx = \frac{h}{3}(y_0 + 4y_1 + y_2) - \frac{h^5}{90}f^{(4)}(\xi),$$

kjer je $\xi \in [x_0, x_2]$.Pri Simpsonovem pravilu funkcijo interpoliramo s parabolo in integral parabole vzamemo za približek integrala funkcije. Tudi tu smo uteži

$$A_0 = \int_{x_0}^{x_2} \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)}\,dx = \frac{h}{3},$$
$$A_1 = \int_{x_0}^{x_2} \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)}\,dx = \frac{4h}{3},$$
$$A_2 = \int_{x_0}^{x_2} \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}\,dx = \frac{h}{3}$$

določili z integriranjem Lagrangeeovih baznih polinomov.



Slika 7.1: Integracija funkcije s trapeznim pravilom.

Za grafično primerjavo obeh pravil pri integraciji funkcije $f(x) = x^2 - x + 2 + 3\sin 2x \cos x$ si poglejmo sliki 7.1 in 7.2.

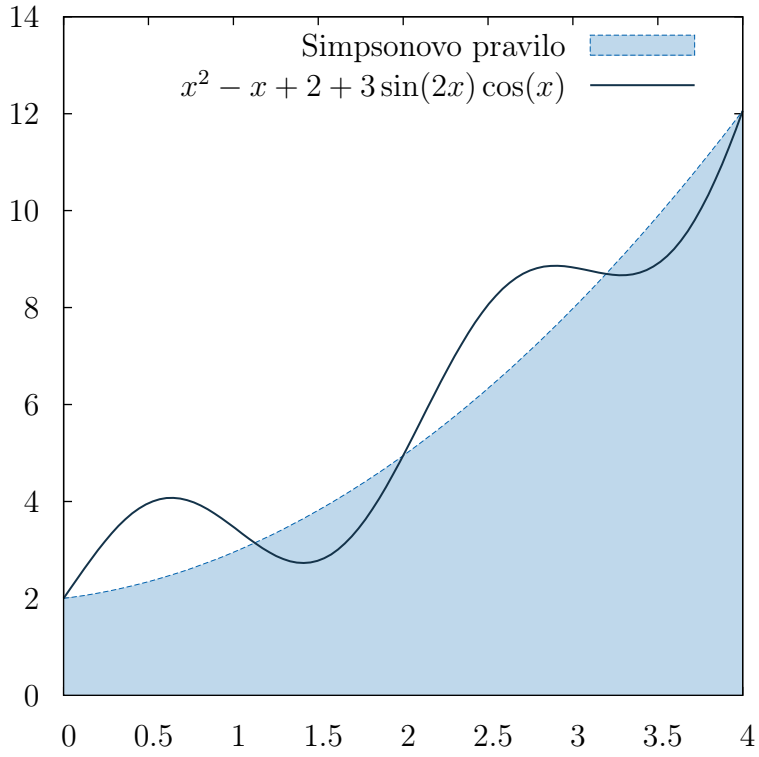
7.1.2 Sestavljeno Simpsonovo pravilo

Zaradi neodstranljive napake namesto večanja stopnje integracijskega pravila raje razdelimo integracijski interval na manjše podintervale. Pri tem se naslonimo na naslednjo trditev.

Trditev 7.0.2 *Če je f definirana in zvezna na intervalu $[a, b]$, razen v mogoče končnem številu točk, potem velja*

$$\int_a^b f(x)\,dx = \int_a^c f(x)\,dx + \int_c^b f(x)\,dx$$

za poljubni števili a in b in število $c \in [a, b]$.



Slika 7.2: Integracija funkcije s Simpsonovim pravilom.

Na ta način dobimo sestavljena integracijska pravila. Naj bo n sodo naravno število. Interval $[a, b]$ razdelimo z ekvidistantnimi točkami x_i:

$$h = \frac{b-a}{n},$$
$$x_i = a + ih; \quad i = 0, 1, \cdots, n.$$

Očitno je $a = x_0$ in $b = x_n$.

Imejmo $\frac{n}{2}$ disjunktnih intervalov $[x_{2i}, x_{2i+2})$ za $i = 0, 1, \ldots, \frac{n}{2} - 1$. Če na vsakem takem intervalu uporabimo običajno Simpsonovo pravilo in te integrale seštejemo, dobimo:

$$\begin{aligned} \int_{x_0}^{x_n} f(x)\,dx &= \sum_{i=0}^{n/2-1} \int_{x_{2i}}^{x_{2i+2}} f(x)\,dx = \\ &= \sum_{i=0}^{n/2-1} \left(\frac{h}{3}(y_{2i} + 4y_{2i+1} + y_{2i+2}) - \frac{h^5}{90} f^{(4)}(\xi_i) \right) \end{aligned} \tag{7.1}$$

kjer je lokalna napaka $\xi_i \in [x_{2i}, x_{2i+2}]$. Zaradi seštevanja $\Theta(n)$ lokalnih napak, je globalna napaka za en red manjša. Po preoblikovanju izraza 7.1 dobimo, da je

$$\int_a^b f(x)\,dx \approx \frac{h}{3} \left(y_0 + 2 \sum_{i=1}^{n/2-1} y_{2i} + 4 \sum_{i=1}^{n/2} y_{2i-1} + y_n \right) \tag{7.2}$$

aproksimacija integrala s sestavljenim Simpsonovim pravilom.

7.1.3 Večdimenzionalni integrali

Računamo integral funkcije $f(x_1, x_2, \ldots, x_d)$ z d spremenljivkami po kvadru $\Omega = [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$. Če vsakega izmed intervalov $[a_i, b_i]$ razdelimo ekvidistantno z $n_i + 1$ točkami, je razdalja med posameznima točkama $h_i = \frac{b_i - a_i}{n_i}$. Potem velja

$$\int_\Omega f\,d\Omega \approx \frac{h_1 \cdots h_d}{3^d} \sum_{i_1=0}^{n_1} \cdots \sum_{i_d=0}^{n_d} A(i_1, n_1) \cdots A(i_d, n_d) f(a_1 + i_1 h_1, \ldots, a_d + i_d h_d),$$

kjer je

$$A(i,n) = \begin{cases} 1; & i \in \{0, n\} \\ 4; & i \in \{1, 3, \ldots, n-1\} \\ 2; & \text{sicer} \end{cases}$$

Težava pri računanju integrala funkcije več spremenljivk je, da že pri malem številu delilnih točk n_i potrebujemo $\Theta(n_1 n_2 \cdots n_d)$ evaluacij funkcije, kjer je d število spremenljivk oz. dimenzija integrala. Zaradi tega se pri velikih d bolje obnese metoda Monte Carlo.

7.1.4 Metoda Monte Carlo

Integral

$$I(f) = \frac{1}{b-a} \int_a^b f(x)\, dx$$

je enak povprečni vrednosti funkcije f na intervalu $[a, b]$. Naj bo X slučajna spremenljivka, enakomerno porazdeljena po na intervalu $[a, b]$. Približek za integral je

$$\int_a^b f(x) \approx \frac{b-a}{n} \sum_{i=1}^{n} f(X_i),$$

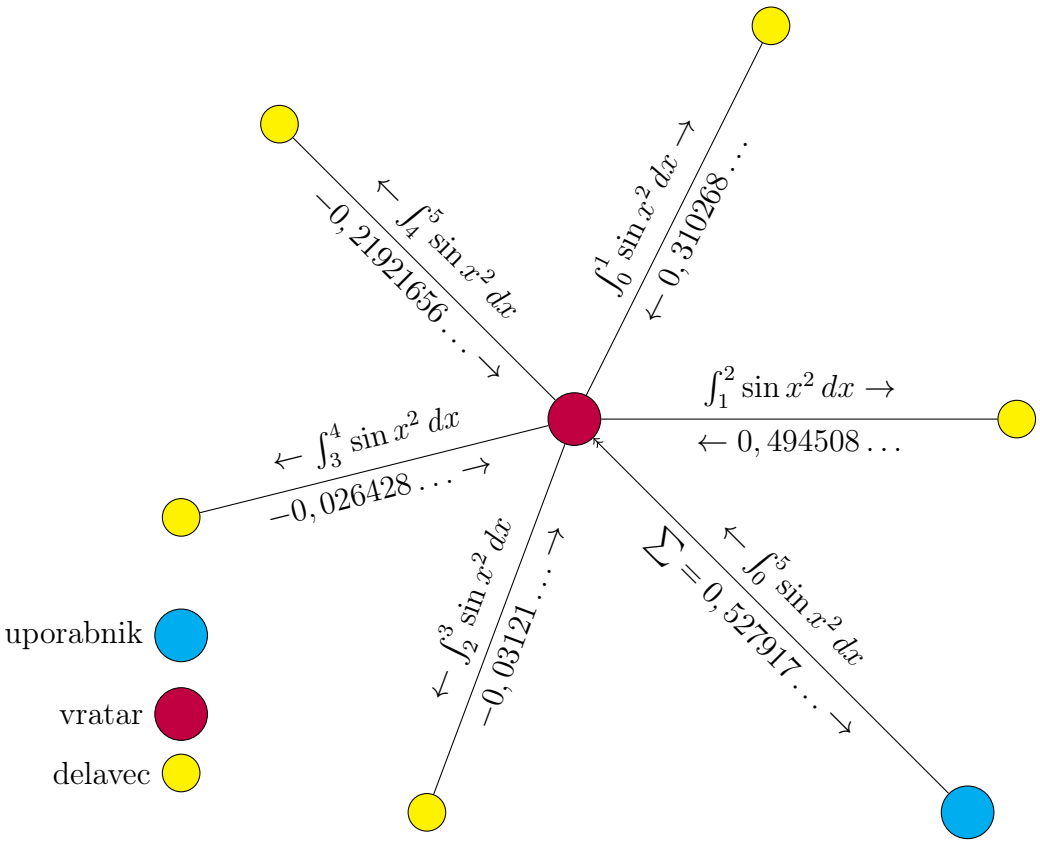
kjer so X_i neodvisne naključne vrednosti slučajne spremenljivke X.

Metoda Monte Carlo deluje na verjetnostnem modelu, zato potrebujemo hiter postopek, kako generirati dovolj dobra psevdonaključna števila.

7.2 Topologija omrežja

Zaradi trditve 7.0.2 si lahko dovolimo, da integracijsko območje razdelimo na poljubno mnogo podobmočij. Ta podobmočja lahko v želenih razmerjih razdelimo med razpoložljive računske vire, na koncu pa vse rezultate seštejemo.

Zaradi enostavnosti izračuna končne vsote v primerjavi s celotnim delom, lahko rečemo, da je problem numeričnega integriranja *prijetno paralelen.* Za reševanje tega problema je najbolj ustrezna topologija zvezde (slika 7.3), lahko pa uporabimo tudi topologijo drevesa z visoko stopnjo vejanja.



Slika 7.3: Aktorje razporedimo v obliki zvezde. Uporabnik izda povpraševanje vratarju za izračun integrala $\int_0^5 \sin x^2\,dx$, vratar pa delo naprej razdeli med delavce in sproti sešteva njihove odgovore. Ko končajo vsi delavci, odgovori uporabniku z rezultatom.

7.3 Izvedba algoritma

7.3.1 Sporočila med aktorji

Pri izdelavi arhitekture v aktorskem modelu lahko začnemo z razmišljanjem o tem, kakšne vrste sporočil si bodo aktorji med seboj pošiljali.

V našem primeru bomo potrebovali tri različne vrste sporočil.

- Prvi tip sporočila (`MonteCarlo(obmocje, f, iteracij)`), bo zahtevek

delavcu, naj izračuna integral po metodi Monte Carlo. Skupaj s takim sporočilom bo prispelo še območje, poljuben kvader v $\mathbb{R}^d$, funkcija $f : \mathbb{R}^d \to \mathbb{R}$ ter želeno število iteracij.

- Drug tip sporočila (`Simpson(obmocje, f, delitev)`) naroči delavcu, naj izračuna določeni integral po Simpsonovi metodi. Parametri takega sporočila so popolnoma enaki kot prej, le da imamo namesto števila iteracij podan vektor v $\mathbb{N}^d$, ki nam pove število delilnih točk v posamezni dimenziji.

- Tretji tip sporočila (`Rezultat(rezultat)`) pa je zgolj odgovor delavca svojemu nadzorniku, da je končal z izračunom. V vsebini tega sporočila se nahaja rezultat numerične integracije.

Vse tri vrste sporočil so jasno prikazane v programski kodi 7.

```
object Integracija {
   case class MonteCarlo(
      obmocje: Array[(Double, Double)],
      f: Array[Double] => Double,
      iteracij: Int
   )
   case class Simpson(
      obmocje: Array[(Double, Double)],
      f: Array[Double] => Double,
      delitev: Array[Int]
   )
   case class Rezultat(
      rezultat: Double
   )
}
```

Programska koda 7: Aktorji si med seboj pošiljajo tri različna sporočila: zahtevek za izračun integrala po Simpsonovi metodi ter metodi Monte Carlo in sporočilo, ki vsebuje rezultat izračuna.

7.3.2 Aktor delavec

Sedaj ko smo popolnoma definirali vsa sporočila pa je čas, da napišemo definicijo prvega aktorja. Poimenovali ga bomo `Delavec`. Aktorju moramo definirati metodo `receive`, v kateri napišemo navodila, kako naj aktor ravna, ko prejme določeno sporočilo.

Najprej si poglejmo prejem sporočila `MonteCarlo(obmocje, f, iteracij)`. Za pomoč pri razvoju algoritma smo definirali funkcijo, ki vrne naključno število na intervalu $[zacetek, konec)$. To funkcijo pokličemo za vsako stranico integracijskega kvadra enkrat, kar nam da naključno točko v tem kvadru. V tej točki izračunamo funkcijsko vrednost. To ponavljamo tolikokrat, kot nam je podano v spremenljivki `iteracij`.

Na koncu je treba poskrbeti, da pošiljatelju sporočila odgovorimo s povprečjem vseh funkcijskih vrednosti. To storimo tako, da pokličemo metodo `!` na objektu `sender`, ki označuje pošiljatelja trenutnega sporočila.

Končna programska koda 8 je enostavna in hitro razumljiva.

```
1  class Delavec extends Actor {
2     def receive = {
3        case MonteCarlo(obmocje, f, iteracij) =>
4           def nakljucnoStevilo(zacetek: Double, konec: Double) =
5              zacetek + (konec - zacetek) * util.Random.nextDouble
6
7           var vsota = 0.0
8           for (i <- 1 to iteracij) {
9              val tocka = obmocje.map { case (zacetek, konec) =>
10                nakljucnoStevilo(zacetek, konec)
11             }
12             vsota += f(tocka)
13          }
14          sender ! Rezultat(vsota / iteracij)
15    }
16 }
```

Programska koda 8: Algoritem za izvedbo numerične integracije po poljubnem kvadru v $\mathbb{R}^d$ z uporabo metode Monte Carlo.

Poglejmo si še izvedbo Simpsonove metode v poljubni dimenziji. Prva stvar, ki jo naredimo ob prejemu sporočila je, da preverimo, če je na vsaki stranici integracijskega kvadra sodo število delilnih točk, saj je to ena izmed zahtev algoritma. Definiramo si tudi spremenljivko `vsota` in ugotovimo dimenzijo itegracijskega kvadra.

Ker imamo problem v dinamičnem številu dimenzij, moramo postopati rekurzivno. Začnemo s praznim vektorjem in koeficientom 1 (nevtralnim elementom za množenje). V vektor rekurzivno dodamo vse mogoče vozle v trenutni dimenziji in pomnožimo koeficient z 1 (vozel v kotu), 4 (lih vozel) ali 2 (sod vozel). Ko je dolžina vektorja enaka dimenziji, prištejemo vrednost funkcije v tem vektorju, pomnožene s končnim koeficientom.

Končno vsoto pomnožimo z faktorjem $\frac{h_1 h_2 \ldots h_d}{3^d}$, kjer je d dimenzija kvadra, h_i pa razdalja med zaporednima vozloma v i-ti dimenziji.

```
1  class Delavec extends Actor {
2     def receive = {
3        case Simpson(obmocje, f, delitev) =>
4           require(delitev.forall(_ % 2 == 0),
5              "Na vsaki stranici kvadra naj bo sodo število delilnih točk"
6           )
7
8           var vsota = 0.0
9           val d = obmocje.size
10
11          def generirajTocke(
12             ostalo: Int = d,
13             koeficient: Double = 1.0,
14             tocka: List[Double] = List()
15          ): Unit =
16             if (ostalo == 0) {
17                vsota += koeficient * f(tocka.toArray)
18             } else {
```

```
19                val (zacetek, konec) = obmocje(ostalo-1)
20                val n = delitev(ostalo-1)
21                for (i <- 0 to n) {
22                   val koordinata = zacetek + (konec - zacetek) * i / n
23                   val novKoef = koeficient * if (i % 2 == 1) 4
24                      else if (i == 0 || i == n) 1 else 2
25                   generirajTocke(ostalo - 1, novKoef, koordinata :: tocka)
26                }
27             }
28
29          generirajTocke()
30          val faktor = (obmocje, delitev).zipped.map {
31             case ((zacetek, konec), n) => (konec - zacetek) / n
32          }.product / Math.pow(3.0, d)
33          sender ! Rezultat(faktor * vsota)
34    }
35 }
```

Programska koda 9: Algoritem za izvedbo numerične integracije po poljubnem kvadru v $\mathbb{R}^d$ z uporabo Simpsonove metode.

Mogoče izboljšave

Naša izvedba Simpsonovega algoritma je zaradi svoje splošnosti nekoliko počasnejša pri integriranju funkcije z malo spremenljivkami. To pomanjkljivost bi lahko odpravili tako, da bi napisali specializirane algoritme za prvih nekaj dimenzij, na primer $f(x)$, $f(x, y)$ in $f(x, y, z)$. Vsaj funkcije ene spremenljivke bi lahko pohitrili z uporabo katere izmed *adaptivnih metod*.

7.4 Programski vmesnik (API)

Če želi uporabnik uporabljati našo knjižnico ali pa zgolj našo oblačno storitev, lahko povsem enostavno pošlje vratarju eno izmed sporočil `MonteCarlo(obmocje, f, iteracij)` ali `Simpson(obmocje, f, delitev)` in

počaka na odgovor. Tak način uporabe sicer ni težek, je pa res, da ni zelo praktičen. S tem namenom smo razvili programski vmesnik, kjer za pošiljanje teh sporočil namesto uporabnika skrbijo priročne funkcije.

Programski vmesnik vsebuje samo eno metodo, `integrate`, kateri podamo d parov (a_i, b_i), ki predstavljajo integracijski kvader, in (parcialno) funkcijo $f : \mathbb{R}^d \to \mathbb{R}$.

Enostavni integral

Začnimo z enostavnim primerom, integriranjem funkcije ene spremenljivke. Integral

$$\int_2^3 \sin x^2 \, dx$$

lahko numerično izračunamo z uporabo spodnjega koščka kode:

```
// Integrira funkcijo sin x^2 na intervalu [2,3]
integrate( (2,3) )( case Array(x) =>
   sin(pow(x, 2))
)
```

Mnogoterni integral

Če imamo funkcijo več spremenljivk in njen integral

$$\int_0^1 \int_{-10}^{10} \sin x^2 \cos y \, dy \, dx,$$

potem je uporaba povsem enaka, le integracijskemu kvadru je potrebno dodati še eno dimenzijo in funkciji en parameter:

```
integrate( (0,1), (-10, 10) ) { case Array(x, y) =>
   sin(x*x) * cos(y)
}
```

Integral po poljubnem območju

Recimo, da želimo integrirati funkcijo $f(x) = \sin x \cos y \tan z$ po enotski krogli s središčem v koodrinatnem izhodišču:

$$\iiint_{x^2+y^2+z^2 \leq 1} \sin x \cos y \tan z \, dx \, dy \, dz \, .$$

V tem primeru pogonu v prvem parametru povemo, v katerem kvadru je vsebovano celotno integracijsko območje. V našem primeru je to kar kocka $[-1, 1]^3$. V drugem parametru pa namesto totalne funkcije podamo parcialno funkcijo, torej tako, ki ni definirana na celotnem območju prej omenjenega vsebovalnega kvadra:

```
integrate( (-1,1), (-1,1), (-1,1) ) {
   case Array(x, y, z) if (sqrt(x*x + y*y + z*z) < 1) =>
      sin(x) * cos(y) * tan(z)
}
```

V uporabo parcialnih funkcij nismo prisiljeni, možnost imamo namreč podati totalno funkcijo, ki na nedefiniranem območju vrača 0:

```
integrate( (-1,1), (-1,1), (-1,1) ) { case Array(x, y, z) =>
   if (sqrt(x*x + y*y + z*z) < 1)
      sin(x) * cos(y) * tan(z)
   else 0
}
```

Poglavje 8

Množenje matrik

8.1 Topologija omrežja

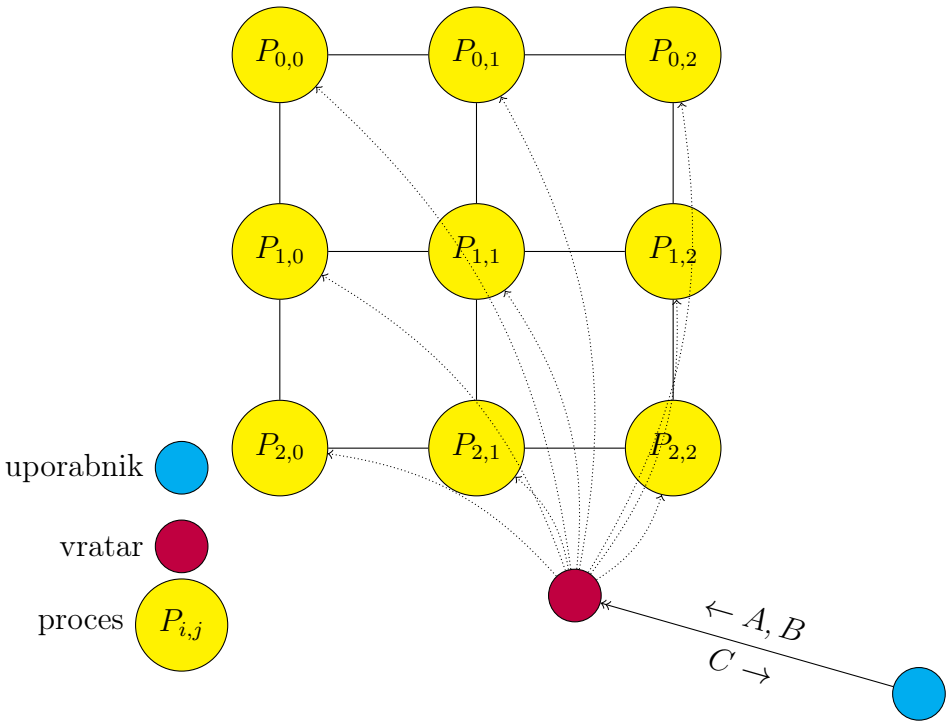
Imejmo p^2 procesov, razporejenih v kvadratno mrežo dimenzije $p \times p$, kot je to prikazano na sliki 8.1 za $p = 3$. Očitno zahtevamo, da skupno število procesov popoln kvadrat, kar je lahko precejšnja omejitev, če ne gradimo fizičnega sistema s to zahtevo v mislih. Označimo proces v $(i \in \mathbb{N}_0)$-ti vrstici in $(j \in \mathbb{N}_0)$-tem stolpcu z oznako $P_{i,j}$. Proces $P_{i,j}$ bo svoje začetne podatke vedno prejel od vratarja, nato pa bo komuniciral samo še z ostalimi procesi, le končni rezultat bo spet sporočil nazaj vratarju.

Želimo izračunati produkt

$$C = A \cdot B,$$

kjer je A matrika velikosti $n \times k$, B matrika velikosti $k \times m$ in C matrika velikosti $n \times m$. Za osnovni algoritem zahtevamo, da so števila n, k in m deljiva s p. Če niso, moramo matrike pred uporabo algoritma razširiti z ničelnimi elementi do velikosti najbližjega večkratnika. Ker je $p \ll n, k, m$, to pomeni, da taka razširitev ne vpliva znatno na učinkovitost algoritma.

Ko so enkrat vse tri matrične dimenzije deljive s p, lahko vsako izmed matrik A, B in C razrežemo na $p \times p$ blokov. Naj notacija $M(i, j)$ predstavlja blok v $(i \in \mathbb{N}_0)$-ti vrstici in $(j \in \mathbb{N}_0)$-tem stolpcu matrike M.



Slika 8.1: Procese razporedimo v obliki mreže. Uporabnik izda povpraševanje vratarju za množenje matrik A in B, vratar pa razdeli bloke teh dveh matrik med procese v mreži. Ti med sabo komunicirajo na zahtevan način, nato pa vratar spet zbere njihove rezultate in jih posreduje uporabniku.

Zadolžimo proces $P_{i,j}$ za računanje bloka $C(i,j)$. Na ta način bo imel vratar na koncu zelo lahko delo, saj bo na enostaven način bločno matriko C spet sestavil nazaj iz odgovorov delavcev.

8.2 Cannonov algoritem

Naša želja in osnovna ideja Cannonovega algoritma je, da vsakemu izmed procesov dodelimo en blok matrike A in en blok matrike B tako, da bo lahko že takoj opravil prvo bločno množenje blokov $A(i,k)$ in $B(k,j)$, ki jih je prejel. To lahko storimo tako, da blok $A(i,j)$ dodelimo procesu $P_{i,j-i \pmod p}$, blok

$A(0,0)$, $B(0,0)$	$A(0,1)$, $B(1,1)$	$A(0,2)$, $B(2,2)$
$A(1,1)$, $B(1,0)$	$A(1,2)$, $B(2,1)$	$A(1,0)$, $B(0,2)$
$A(2,2)$, $B(2,0)$	$A(2,0)$, $B(0,1)$	$A(2,1)$, $B(1,2)$

Tabela 8.1: Začetna razdelitev matrik A in B v 3×3 mrežo procesov $P_{i,j}$, ki omogoča že prvo bločno množenje. Blok $A(i,j)$ smo dodelili procesu $P_{i,j-i \pmod p}$, blok $B(i,j)$ pa procesu $P_{i-j \pmod p,j}$.

$A(0,1)$, $B(1,0)$	$A(0,2)$, $B(2,1)$	$A(0,0)$, $B(0,2)$
$A(1,2)$, $B(2,0)$	$A(1,0)$, $B(0,1)$	$A(1,1)$, $B(1,2)$
$A(2,0)$, $B(0,0)$	$A(2,1)$, $B(1,1)$	$A(2,2)$, $B(2,2)$

Tabela 8.2: Razdelitev matrik A in B po koncu prvega koraka. Vsak proces je posredoval svoj blok matrike A ciklično levo ter svoj blok matrike B ciklično navzgor.

$B(i,j)$ pa procesu $P_{i-j \pmod p,j}$. S tem smo vsak blok vsake matrike dodelili natanko enemu procesu. Primer delitve za mrežo procesov 3×3 si lahko pogledamo v tabeli 8.1.

Ideja algoritma je taka, da so procesi med seboj sposobni posredovati bloke matrik A in B drug drugemu. Če se lahko domislimo pametnega pravila za tako komunikacijo, se bo algoritem ustavil po p korakih, kar je idealno. Na srečo tako pravilo obstaja in je tudi precej enostavno. Vsak proces $P_{i,j}$ posreduje svoj blok matrike A ciklično levo, torej procesu $P_{i,j-1 \pmod p}$ ter svoj blok matrike B ciklično levo, torej procesu $P_{i-1 \pmod p,j}$.

Opazili smo torej, da proces sploh ne potrebuje vseh štirih povezav, ki so mu na voljo v mreži, temveč zgolj dve – navzgor in levo (ciklično v primeru procesov v prvi vrsti in stolpcu).

Kot smo omenili v uvodu, je zelo koristno, da se komunikacija vedno prekriva z izračunavanjem – to lahko storimo tako, da proces posreduje blok še preden začne z množenjem. To posledično pomeni, da vsak izmed procesov potrebuje za en blok več pomnilnika.

8.2.1 Množenje redkih matrik

Redka matrika je taka matrika, katere večina elementov je enakih nič. To nam omogoča, da matriko zapišemo v bolj učinkoviti obliki, kar zmanjša potrebe po pomnilniku in kar je mogoče še bolj pomembno – po komunikaciji, saj je potrebno prenesti dosti manj podatkov.

Zgoraj opisan algoritem, povsem nespremenjen, deluje tudi za redke matrike, če jih zna vsak izmed procesov lokalno množiti in seštevati, kar drži v primeru tehnologij, ki smo jih izbrali mi.

8.2.2 Povzetek algoritma

Naj še enkrat kratko in jedrnato zapišemo celoten algoritem.

Algoritem, ki ga izvaja vratar, je sledeč:

1. Vsakemu procesu $P_{i,j}$ v mreži sporoči naslov zgornjega soseda $P_{i-1 \pmod p,j}$ in levega soseda $P_{i,j-1 \pmod p}$ (ciklično).
2. Pošlje vsak blok $A(i,j)$, procesu v i-ti vrsti in $(j - i \pmod p)$-tem stolpcu.
3. Pošlje vsak blok $B(i,j)$, procesu v $(i - j \pmod p)$-ti vrsti in j-tem stolpcu.
4. Počaka na odgovore vseh procesov in bločno sestavi rezultat, matriko C.

Algoritem, ki ga izvaja vsak proces $P_{i,j}$ je sledeč:

1. Ustvari svoj blok matrike C, ki ga inicializira z ničlami.
2. Za vsak korak $k = 0, 1, \ldots, p-1$:
   (a) Počaka na prejem bloka matrike A in B za k-ti korak.
   (b) Ta dva bloka kar takoj pošlje naprej – levemu sosedu blok A in desnemu blok B.
   (c) Posodobi svoj blok matrike: $C \leftarrow C + A \cdot B$.
3. Pošlje svoj blok matrike C nazaj vratarju.

8.3 Izvedba algoritma

8.3.1 Sporočila med aktorji

Tako kot smo to storili pri numerični integraciji, si najprej definirajmo različne vrste sporočil, ki si jih bodo med seboj pošiljali aktorji.

Prej smo omenili, da bo vsak aktor komuniciral le z aktorjem levo in navzgor od njega, čisto na koncu, po `korakov` korakih, pa bo poslal končen rezultat nazaj vratarju. Z namenom, da lahko posredujemo aktorjem te podatke, smo ustvarili sporočila `Levo`, `Gor` in `Vratar`.

Naprej imamo sporočilo `Mnozi` z dvema matrikama A in B, ki ga prejme vratar od uporabnika. Sporočilo `Rezultat` služi za posredovanje matrike C ali pa samo njenega bloka.

Preostane nam samo še definicija sporočil, s katerimi si procesi posredujejo bloke matrik med sabo. Temu služita sporočili `MatrikaA` in `MatrikaB`, vsako pa vsebuje blok te matrike in številko koraka (od 0 do $p-1$).

```
1  object Mnozenje {
2    case class Levo(aktor: ActorRef)
3    case class Gor(aktor: ActorRef)
4    case class Vratar(aktor: ActorRef, korakov: Int)
5
6    type DMatrix = Matrix[Double]
7
8    case class Mnozi(a: DMatrix, b: DMatrix)
9    case class Rezultat(c: DMatrix)
10
11   case class MatrikaA(a: DMatrix, korak: Int)
12   case class MatrikaB(b: DMatrix, korak: Int)
13 }
```

Programska koda 10: Vse vrste sporočil, ki jih potrebujemo za izvedbo distribuiranega matričnega množenja s Canonnovim algoritmom.

8.3.2 Aktor vratar

Naprej si poglejmo definicijo vratarja za matrično množenje. Ob zagonu ustvari mrežo procesov predpisane dimenzije (v praksi bo ta mreža že na voljo). Vsakemu procesu pove, kdo je vratar (on sam) ter kdo se nahaja navzgor in levo od njega.

Aktor prejema dve vrsti sporočil.

Prvi tip sporočila je `Mnozi`, ki ga prejme od uporabnika, vsebuje pa matriki A in B. Aktor vsako izmed matrik razdeli med prej ustvarjene procese po prej povedanem pravilu.

Drugi tip sporočila pa je `Rezultat`, ki mu ga pošlje vsak izmed procesov po končanih p korakih. To programsko kodo smo zaradi preglednosti izpustili.

```
1  class MnozenjeVratar extends Actor {
2     val p = 3
3
4     // ustvari 3x3 mrežo aktorjev
5     val aktorji = (0 until p).map { i =>
6        (0 until p).map { j =>
7           context.actorOf(Props[MnozenjeDelavec], s"$i,$j")
8        }
9     }
10
11    // vsakemu aktorju podaj sebe ter gornjega in levega soseda
12    for {
13       i <- 0 until p // vrstica
14       j <- 0 until p // stolpec
15       aktor = aktorji(i)(j)
16    } {
17       aktor ! Gor( aktorji((i-1) mod p)(j) )
18       aktor ! Levo( aktorji(i)((j-1) mod p) )
19       aktor ! Vratar(self, p)
20    }
21
22    def receive = {
```

```
23        case Mnozi(a, b) =>
24           // bločno razdeli matriki A in B med delavce
25           a.subdivide(p) { (i, j, m) =>
26              aktorji(i)((j-i) mod p) ! MatrikaA(m, 0)
27           }
28           b.subdivide(k) { (i, j, m) =>
29              aktorji((i-j) mod p)(j) ! MatrikaB(m, 0)
30           }
31        case Rezultat(c) =>
32           // prejmi rezultat od vsakega delavca posebej
33     }
34  }
```

Programska koda 11: Aktor vratar, čigar programska koda je precej enostavna.

8.3.3 Aktor delavec (proces)

Aktor čisto na začetku življenja prejme tri reference v sporočilih `Levo`, `Gor` in `Vratar`, ki si jih zapomni za pozneje.

Od vratarja in ostalih procesov pa prejema bloke matrik A in B, ki jih v vsakem koraku prišteje bloku matrike C, za katerega je zadolžen. Naj še enkrat poudarimo, da se za bločno množenje uporablja zelo učinkovita knjižnica OpenBLAS, saj je to najbolj računsko zahtevna operacija celotnega algoritma.

```
1  class MnozenjeDelavec extends Actor {
2     // tu hranimo reference na tri pomembne aktorje
3     var gor = Option.empty[ActorRef]
4     var levo = Option.empty[ActorRef]
5     var vratar = Option.empty[ActorRef]
6
7     // število množenj, ki jih moramo opraviti
8     var potrebnihKorakov = Option.empty[Int]
9
```

```
10    // koščki matrike, ki smo jih že prejeli, ne pa še obdelali
11    var A = Map.empty[Int, DMatrix]
12    var B = Map.empty[Int, DMatrix]
13
14    // matrika rezultatov in trenutni korak
15    var C: DMatrix = null
16    var korak = 0
17
18    def mnozi {
19       for {
20          g <- gor
21          l <- levo
22          v <- vratar
23
24          a <- A.get(korak)
25          b <- B.get(korak)
26       } {
27          A -= korak
28          B -= korak
29
30          if (korak < potrebnihKorakov.get) {
31             l ! MatrikaA(a, korak+1)
32             g ! MatrikaB(b, korak+1)
33          }
34
35          val blok = a*b
36          if (korak == 0)
37             C = blok
38          else
39             C += blok
40
41          korak += 1
42          if (korak == potrebnihKorakov.get)
43             v ! Rezultat(C)
44       }
45    }
```

```
46
47    def receive = {
48       case Gor(aktor) =>
49          gor = Some(aktor)
50       case Levo(aktor) =>
51          levo = Some(aktor)
52       case Vratar(aktor, korakov) =>
53          vratar = Some(aktor)
54          potrebnihKorakov = Some(korakov)
55
56       case MatrikaA(matrika, korak) =>
57          A += korak -> matrika
58          mnozi
59       case MatrikaB(matrika, korak) =>
60          B += korak -> matrika
61          mnozi
62    }
63 }
```

Programska koda 12: Aktor delavec. Njegova glavna naloga je bločno množenje prejetih matrik.

Poglavje 9

Reševanje sistema linearnih enačb

9.1 Matematično ozadje

Metode za reševanje sistema linearnih enačb v splošnem delimo v dve skupini: direktne metode in iterativne (nedirektne) metode. Pri uporabi direktnih metod že od vsega začetka točno vemo, koliko operacij bomo morali izvesti, da pridemo do rešitve. Take metode dajo točen odgovor samo, če jih izvajamo v aritmetiki z neskončno natančnostjo. Praktično pa se lahko zgodi, da zaradi akumulacije zaokroževalnih napak dajo povsem nesmiselne rezultate.

Za razliko od direktnih metod pa iterativne metode trajajo v neskončnost, na nas pa je, da se odločimo, kdaj smo z rezultatom zadovoljni in prekinemo izračunavanje. Pri tem moramo biti pazljivi, saj se lahko zgodi, da metoda divergira.

Iterativne metode so še posebej uporabne, kadar rešujemo sisteme, pri katerih je matrika A diagonalno dominantna, saj to pomeni hitro konvergenco. Skoraj nujne pa so tudi, kadar je matrika redka, saj iterativni algoritmi pogosto ne vsebujejo matričnih operacij, ki bi povzročile, da bi bil rezultat operacije nad redko matriko gosta matrika. Ravno to je razlog, da se bomo v

tem diplomskem delu posvetili zgolj iterativnim metodam, saj želimo, da vsi opisani algoritmi delujejo tudi na redkih matrikah.

Imejmo spodnji sistem linearnih enačb $Ax = b$, zapisan v matrični obliki:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}, \tag{9.1}$$

kjer je $A = [a_{i,j} \in \mathbb{R}]$ nesingularna $n \times n$ matrika z neničelninimi elementi na diagonali, b pa je vektor desnih strani enačb. Matriko A imenujemo matrika koeficientov.

9.1.1 Gauss-Seidelova iteracija

Za reševanje sistema (9.1) bomo najprej uporabili Gauss-Seidelovo iteracijo. Za namene analize tokrat privzemimo, da so vsi diagonalni elementi $a_{i,i} = 1$, kar lahko dosežemo z deljenjem vsake i-te enačbe z diagonalnim elementom $a_{i,i}$ ([KKN11]). Sedaj zapišimo

$$A = I + L + U, \tag{9.2}$$

kjer je I identična matrika, L spodnje trikotna matrika z ničelnimi elementi na diagonali, U pa zgornje trikotna matrika, prav tako z ničelnimi elementi na diagonali.

Vstavimo sedaj enakost (9.2) v osnovno enačbo $Ax = b$ in preoblikujmo:

$$(I + L + U)x = b$$

$$x + Lx + Ux = b$$

$$x = b - Lx - Ux. \tag{9.3}$$

Če sedaj razpišemo elemente vektorja x iz enačbe (9.3) po elementih, dobimo

$$x_i = b_i - \sum_{j=1}^{i-1} a_{i,j}x_j - \sum_{j=i+1}^{n} a_{i,j}x_j$$

za vsak $i = 1, 2, \ldots, n$ po vrsti, enega za drugim. Sedaj lahko opazimo, da imamo zato v tekoči iteraciji pri računanju elementa x_i že na voljo na novo izračunane elemente $x_1, x_2, \ldots, x_{i-1}$, zato lahko te že nemudoma uporabimo.

Označimo z $x_i^{(k+1)}$ elemente trenutne iteracije, z $x_i^{(k)}$ pa elemente prejšnje. Če sedaj upoštevamo še to, da smo na začetku delili enačbe z diagonalnimi elementi, se Gauss-Seidelova iteracija za splošno matriko $[a_{i,j}]$ glasi:

$$x_i^{(k+1)} = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{i,j} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n. \quad (9.4)$$

9.1.2 Jacobijeva iteracija

Zapišimo matriko A sistema (9.1) kot vsoto

$$A = D + R \quad (9.5)$$

kjer je D diagonalna matrika, R pa je enaka vsoti $L + U$. Če vstavimo (9.5) v $Ax = b$ in izraz preoblikujemo, nam to prinese:

$$(D + R)x = b$$

$$Dx + Rx = b$$

$$x = D^{-1}(b - Rx). \quad (9.6)$$

Podobno kot smo to storili pri Gauss-Seidelovi iteraciji, pa lahko vektor x iz enačbe (9.6) zapišemo po komponentah:

$$x_i^{(k+1)} = \frac{1}{a_{i,i}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{i,j} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n, \quad (9.7)$$

kjer smo upoštevali trivialno dejstvo, da je inverz diagonalne matrike $[d_{i,i}]^{-1}$ kar enak $[1/d_{i,i}]$.

Opazimo lahko tudi, da lahko shemo Jacobijeve iteracije (9.7) dobimo tako, da pri shemi Gauss-Seidelove iteracije (9.4) namesto elementov trenutne iteracije $x_i^{(k+1)}$ vedno vzamemo elemente prejšnje $x_i^{(k)}$.

9.2 Distribuirana Gauss-Seidelova iteracija

Kot je mogoče zaslediti v [Sha09], je distribuirana izvedba Gauss-Seidelove iteracije (9.4) sicer mogoča, ni pa razširljiva (angl. *scalable*, glej pododsek 2.4.1). Še več, popolnoma odpove že pri res zelo majhnem številu procesov – v članku je konkretno naveden primer reševanja kvadratnega sistema velikosti 24000×24000 elementov. Problem dva procesa rešita v $62,344$ sekundah, trije v $53,969$ sekundah in štirje v $55,954$ sekundah.

Očitno je cena komunikacije in premajhna notranja paralelna narava Gauss-Seidelove iteracije le pretrd oreh za učinkovito in razširljivo distribuirano izvedbo, zato se bomo raje posvetili Jacobijevi iteraciji, saj je s tega vidika neprimerljivo boljša.

9.3 Distribuirana Jacobijeva iteracija

Ker nas pri Jacobijevi iteraciji ne omejuje v pododseku 9.1.1 omenjena zahteva, da moramo elemente trenutne iteracije $x_i^{(k+1)}$ računati po vrsti, si lahko dovolimo, da računanje elementov trenutne iteracije razdelimo med delavce v poljubnem razmerju, če so vsi elementi prejšnje iteracije že znani.

Imejmo p procesov P_i ter delilni vektor $d = (d_1, d_2, \ldots, d_p)$, ki označuje želeno delitev $n \times n$ matrik A, R in D^{-1}, zato mora zanj veljati $d_1 + d_2 + \cdots + d_p = n$. Če sedaj te matrike razdelimo v obeh dimenzijah z uporabo delilnega vektorja d, dobimo bločne matriki velikosti $p \times p$. Naj tako kot v poglavju 8 notacija $M(i, j)$ označuje blok matrike M v $(i \in \mathbb{N})$-ti vrstici in $(j \in \mathbb{N})$-tem stolpcu.

Analogno uporabimo delilni vektor za delitev vektorja d in vektorja x. Tudi tu naj notacija $v(i)$ označuje $(i \in \mathbb{N})$-ti del vektorja v. Tako lahko matrično enačbo (9.6) zapišemo v bločni obliki:

$$x^{(k+1)}(i) = D^{-1}(i,i) \left(b(i) - \sum_{j=1}^{p} R(i,j) x^{(k)}(j) \right) \quad i = 1, 2, \ldots, p. \tag{9.8}$$

Zadolžimo sedaj proces P_i za izračun dela vektorja $x^{(k+1)}(i)$, zato mu moramo čisto na začetku posredovati njegov delček vektorja $b(i)$ ter celotno vrsto blokov $A(i,:)$, iz katerih lahko potem proces sam izlušči matriko $D^{-1}(i,i)$ in vrsto blokov $R(i,:)$. Ko ga izračuna, naj rešitev razpošlje čisto vsem procesom in tudi vratarju. To ni tako potratno, kot se sliši, saj je delček vektorja $x^{(k+1)}(i)$ res majhen v primerjavi z matrikami, vsa pošiljanja pa lahko potekajo vzporedno.

Ker je zelo pomembno, da se računanje in komunikacija čim bolj prekrivata, naj proces začne z računanjem $x^{(k+1)}(i)$ že takoj, ko prejme prvi košček $x^{(k)}(j)$, saj mu vsote v shemi (9.8) ni potrebno računati po vrsti. Ta vektor vsote c, ki se postopoma akumulira, bomo temu primerno poimenovali *akumulator*.

Ko se akumulator c napolni (prejme vse delčke vektorja $x^{(k)}$ prejšnje iteracije), izračunamo $x^{(k+1)}(i) = D^{-1}(i,i)(b(i) - c)$, kar nemudoma razpošljemo. Ta cikel se potem ponavlja v nedogled, dokler ga vratar ne ustavi, ko je zadovoljen z rezultatom.

9.3.1 Povzetek algoritma

Algoritem, ki ga izvaja vratar:

1. Ustvari p procesov in vsakemu izmed njih sporoči sebe in vse ostale procese za namene nadaljnje komunikacije.
2. Vsakemu procesu P_i pošlje njegov del vektorja $b(i)$ in celo vrsto blokov matrike $A(i,:)$.
3. Po vsem tem pa vratar samo še čaka na rešitve in ustavi vse procese, ko je $\|x^{(k+1)} - x^{(k)}\|_2 < \epsilon\|x^{(k+1)}\|_2$ oz. ko preteče preveč korakov.

Algoritem, ki ga izvaja vsak proces P_i:

1. Prejme naslov vratarja in vseh ostalih delavcev in si jih shrani za nadaljnjo komunikacijo.

2. Prejme svoj del vektorja b in bloke $A(i,:)$ iz česar izlušči $D^{-1}(i,i)$ ter bloke $R(i,:)$.
3. Za vsak korak $k = 0, 1 \ldots$:
   (a) Nastavi *akumulator* c na ničelni vektor.
   (b) Vsakič, ko prejme $x^{(k)}(j)$, poveča akumulator: $c \leftarrow c + R(i,j)x^{(k)}(j)$.
   (c) Ko prejme vse delčke vektorja $x^{(k)}$, izračuna $D^{-1}(i,i)(b(i) - c)$ in ta rezultat razpošlje vsem ostalim aktorjem.

9.4 Izvedba algoritma

9.4.1 Sporočila med aktorji

Kot smo sedaj že navajeni, pri izdelavi arhitekture sistema vedno najprej analiziramo vrste sporočil, ki si jih bodo aktorji pošiljali med seboj.

Namen sporočila `Okolica(vratar, okolica)` je, da delavca seznanimo z ostalimi delavci in s seboj. To je pomembno, saj za delovanje algoritma potrebujemo komunikacijo med vsemi delavci. Ni pa to edino sporočilo, ki ga pošljemo delavcu samo enkrat, ob njegovi ustvaritvi. Moramo mu poslati še njegov del vektorja b, kar naredimo s sporočilom `KoscekB(b)`, ter celo vrsto blokov $A(i,j)$, kar naredimo z več sporočili `KoscekA(a, stolpec)`, zato moramo v izogib zmešnjavi povedati tudi številko stolpca j.

Edino sporočilo, ki nam še preostane je `Rezultat(x, odKoga)`, namenjeno pošiljanju vmesnih približkov. Skupaj s sporočilom proces P_i pove tudi svoje število i.

```
1 object ResevanjeSistema {
2   case class Okolica(
3     vratar: ActorRef,
4     okolica: IndexedSeq[ActorRef]
5   )
6
7   type DMatrix = Matrix[Double]
```

```
8     type DVector = DenseVector[Double]
9
10    case class Sistem(a: DMatrix, b: DVector)
11
12    case class KoscekA(a: DMatrix, stolpec: Int)
13    case class KoscekB(b: DVector)
14
15    case class Rezultat(x: DVector, odKoga: Int)
16 }
```

Programska koda 13: Pri reševanju sistema linearnih enačb si aktorji med seboj pošiljajo pet različnih sporočil.

9.4.2 Aktor vratar

Poglejmo si najprej definicijo vratarja. Definirali smo mu tri konstante, in sicer $p = 64$ (število delavcev), $\epsilon = 10^{-13}$ (zadovoljiva razlika kvadratov dveh zaporednih približkov) ter varnostno omejitev števila korakov, v primeru, da iteracija ne konvergira.

Nato ustvarimo p delavcev ter si izdelamo polje, kjer hranimo zadnja dva približka, ki ju je poslal posamezen delavec.

Vratar prejema dve vrsti sporočil:

- Prva vrsta sporočila je ukaz uporabnika `Sistem(a, b)`, da naj vratar in njegovi delavci začnejo z reševanjem linearnega sistema enačb. Skupaj s sporočilom pride matrika A in vektor desnih strani b. Oboje vratar ustrezno razdeli med delavce.

- Druga vrsta sporočila pa je obvestilo posameznega delavca P_i, da je končal z eno iteracijo (`Rezultat(rezultat, odKoga)`). V njem pove rezultat ter svojo zaporedno številko i. Ker hranimo za vsakega delavca zadnja dva približka, si nov približek shranimo ter starejšega zavržemo. Če smo prejeli od vsakogar vsaj dva približka, preverimo resničnost izraza $\|x^{(k+1)} - x^{(k)}\|_2 < \epsilon\|x^{(k+1)}\|_2$. Če je ta resničen ali pa smo že

presegli omejitev števila korakov, pošljemo vsem delavcem tabletko s strupom `PoisonPill` in tako končamo njihovo delo.

```
1  class SistemVratar extends Actor {
2     val p = 64
3
4     val epsilon = 1e-13
5     val omejitevKorakov = 100
6
7     var korak = 1
8
9     // ustvari p delavcev
10    val delavci = (0 until p).map { i =>
11       context.actorOf(Props[SistemDelavec], s"$i")
12    }
13
14    // sporoči vsakemu delavcu sebe in vse ostale
15    delavci.foreach { delavec =>
16       delavec ! Okolica(self, delavci)
17    }
18
19    // od vsakogar zadnja 2 približka
20    var resitve = IndexedSeq.fill(p)(
21       (null: DVector, null: DVector)
22    )
23
24    def receive = {
25       case Sistem(a, b) =>
26          a.subdivide(p) { (i, j, m) =>
27             delavci(i) ! KoscekA(m, j)
28          }
29          b.subdivide(p) { (i, v) =>
30             delavci(i) ! KoscekB(v)
31          }
32
33       case Rezultat(rezultat, odKoga) =>
```

```
34          val prej = resitve(odKoga)._2
35          resitve = resitve.updated(odKoga, (prej, rezultat))
36
37          if (korak > 2*p) {
38             val vsotaKvadratov = resitve.map { case (a, b) =>
39                norm(a - b)
40             }.sum
41             val omejitev = epsilon * resitve.map(_._2.norm).sum
42
43             // če je treba končati, "zastrupi" delavce
44             if (vsotaKvadratov < omejitev || korak > p * omejitevKorakov)
45                delavci.foreach(_ ! PoisonPill)
46          }
47          korak += 1
48    }
49 }
```

Programska koda 14: Vratar je zadolžen za delitev dela in ustavitev delavcev, ko je zadovoljen z rezultatom.

9.4.3 Aktor delavec (proces)

Poglejmo si na kratko še delovanje posameznega delavca P_i (`SistemDelavec`). Odziv na prejem sporočila `Okolica` je trivialen, zato naj si ga bralec sam pogleda v programski kodi 15.

Ob prejemanju blokov matrike A v sporočilu `KoscekA` je delavec sposoben izluščiti diagonalno matriko D ter bloke matrike R. Diagonalno matriko je potrebno pred uporabo še obrniti – zaradi učinkovitosti algoritma pa shranimo njeno diagonalo kot vektor. Ob prejemu sporocila `KoscekB`, ki vsebuje del vektorja b pa začnemo z izračunavanjem, in sicer s približkom $x^{(0)}(i) = (0, 0, \ldots, 0)$.

Delavec P_i ima prej omenjeno spremenljivko `akumulator` ter spremenljivko `preostalih`, ki označuje, koliko približkov $x^{(k)}(j)$ še potrebuje. Kot

ta pade na 0 in je zbral vse približke, se pokliče funkcija `razposlji`, ki stori naslednje:

1. Izračunamo svoj novi približek $x^{(k+1)}(i)$ z uporabo vrednosti akmulatorja. Ta približek takoj razpošljemo vratarju in vsem delavcem.
2. Vektor `akumulator` nastavimo nazaj na ničelni vektor, spremenljivka `preostalih` pa se spet nastavi na p (število delavcev). Tako smo pripravljeni na nov korak Jacobijeve iteracije.

```
1  class SistemDelavec extends Actor {
2     var vratar: ActorRef = null
3     var okolica: IndexedSeq[ActorRef] = null
4     var stevilka = 0
5
6     var bloki = Map.empty[Int, DMatrix]
7     var diagonala: DVector = null
8     var b: DVector = null
9
10    var akumulator: DVector = null
11    var preostalih = -100
12
13    def razposlji {
14       val priblizek = diagonala :* (b - akumulator)
15       vratar ! Rezultat(priblizek, stevilka)
16       okolica.foreach(_ ! Rezultat(priblizek, stevilka))
17
18       akumulator *= 0.0
19       preostalih = okolica.size
20    }
21
22    def receive = {
23       case Okolica(vratar, okolica) =>
24          this.vratar = vratar
25          this.okolica = okolica
26          stevilka = okolica.indexOf(self)
```

```
27            preostalih = okolica.size
28
29         case KoscekA(a, stolpec) =>
30            if (stolpec == stevilka) {
31               diagonala = diag(a).copy.map(1/_)
32               diag(a) *= 0.0
33            }
34            bloki += stolpec -> a
35
36         case KoscekB(v) =>
37            b = v
38            akumulator = DenseVector.zeros(v.size)
39            razposlji
40
41         case Rezultat(x, odKoga) =>
42            preostalih -= 1
43            akumulator += bloki(odKoga) * x
44            if (preostalih == 0)
45               razposlji
46      }
47   }
```

Programska koda 15: Programska logika vsakega delavca.

Poglavje 10

Sklepna beseda

V tem diplomskem delu smo predstavili teoretično ozadje sočasnega programiranja z več različnih vidikov, od motivacije za tak način izračunavanja, vrst sočasnih sistemov, drobitve problema na manjše, učinkovitosti, itd. Opozorili smo tudi na pasti pri izvedbi sistemov z deljenim pomnilnikom. Čisto svoje poglavje smo posvetili distribuiranim sistemom in omrežni komunikaciji. Odločili smo se, da bomo naš distribuiran sistem izvedli s pomočjo aktorskega modela in ogrodja Akka in si tako olajšali delo.

Za programski jezik smo si izbrali Scalo, saj je precej nov in veliko obeta – ponuja pa uglajeno igro objektnega in funkcijskega programiranja. Napisali smo tudi kratek uvod v funkcijsko programiranje.

Vpeljali smo teoretično ozadje numerične integracije v poljubno dimenzijah s pomočjo sestavljenega Simpsonovega pravila in metode Monte Carlo. Za množenje tako gostih kot redkih matrik smo vpeljali Cannonov algoritem, ki smo ga razširili za poljubne matrike (v literaturi se namreč pojavlja zgolj za množenje dveh kvadratnih matrik). Paralelizirali smo tudi Jacobijevo iteracijo za reševanje gostih in redkih sistemov linearnih enačb, algoritem pa nam je uspelo izvesti tako, da se v idealnem primeru čisto vsa komunikacija prekrije z izračunavanjem.

Napisano predstavlja delo študenta matematike in računalništva, zato se nam zdi, da smo dosegli idealen preplet obeh področij. Vsi opisani algoritmi

so bili tudi uspešno izvedeni in praktično preizkušeni, zato ocenjujemo, da smo uspešno izpolnili čisto vse cilje, ki smo si jih zadali.

Literatura

[Mat] Norm Matloff. *Programming on Parallel Machines. GPU, Multicore, Clusters and More*. Angl. University of California, Davis. URL: `http://heather.cs.ucdavis.edu/parprocbook` (pridobljeno 28. 7. 2014).

[KKN11] Erwin Kreyszig, Herbert Kreyszig in E. J. Norminton. *Advanced Engineering Mathematics*. 10. izd. Hoboken, N.J.: Wiley, 2011. ISBN: 0470458364.

[OSV11] Martin Odersky, Lex Spoon in Bill Venners. *Programming in Scala. A Comprehensive Step-by-Step Guide*. Angl. 2. izd. Artima Incorporation, 2011. ISBN: 0981531644, 9780981531649.

[Čer10] Aleš Černivec. "Porazdeljen algoritem za problem najmanjšega k-centra". Magistrska naloga. Univerza v Ljubljani, Fakulteta za računalništvo in informatiko, 2010. Pogl. 1, str. 3–7. 85 str. URL: `http://eprints.fri.uni-lj.si/1261/` (pridobljeno 17. 7. 2014).

[Hew14] Carl Hewitt. "Actor Model for Discretionary, Adaptive Concurrency". Ver. 33. V: *CoRR* abs/1008.1459 (2010, revidirano 2014).

[Ple10] Bor Plestenjak. *Numerične metode 2*. 13. jul. 2010, str. 106–110, 121–123.

[Sha09] Yueqiang Shang. "A distributed memory parallel Gauss–Seidel algorithm for linear algebraic systems". Angl. V: *Computers and Mathematics with Applications* 57.8 (apr. 2009). Ur. Leszek Demkowicz, str. 1369–1376. ISSN: 0898-1221. DOI: 10.1016/j.camwa.2009.01.034.